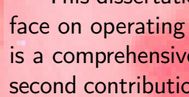
Multi-core chips are currently in the spotlight as a potential means to overcome the limits of frequency scaling for performance increases in processors. In this direction, the CSA group at the University of Amsterdam is investigating a new design for processors towards faster and more efficient general-purpose multi-core chips. However this design changes the interface between the hardware and software, compared to existing chips, in ways that have not been dared previously. Consequently, the concepts underlying existing operating systems and compilers must be adapted before this new design can be fully integrated and evaluated in computing systems.

This dissertation investigates the impact of the changes in the machine interface on operating software and makes four contributions. The first contribution is a comprehensive presentation of the design proposed by the CSA group. The second contribution is formed by technology that demonstrates that the chip can be programmed using standard programming tools. The third contribution is a demonstration that the hardware components can be optimized by starting to implement operating software during the hardware design instead of afterwards. The fourth contribution is an analysis of which parts of the hardware design will require further improvements before it can be fully accepted as a general-purpose platform. The first conclusion is a confirmation that the specific design considered can yield higher performance at lower cost with relatively minimal implementation effort in software. The second conclusion is that the processor interface can be redefined while designing multi-core chips as long as the design work is carried out hand in hand with operating software providers.



UNIVERSITY OF AMSTERDAM

ISBN 978-94-6108-320-3
90000 >
9 789461 083203

On the realizability of hardware microthreading

Raphael 'kena' Poss

Uitnodiging

Voor het bijwonen
van de openbare
verdediging
van mijn proefschrijft:

On the realizability
of hardware microthreading

On the realizability of hardware microthreading

Revisiting the general-purpose processor interface: consequences and challenges

Raphael 'kena' Poss

Woensdag 5 september 2012
om 10:00

Anietenkapel
Oudezijds Voorburgwal 231
1012 EZ Amsterdam

Stellingen

behorende bij het proefschrift

On the realizability
of hardware microthreading

1. General-purpose computers are, like stem cells for living organisms, key to the perpetuation of computer engineering. *(Chapter 1)*

2. Het samenstellen van verwerkingseenheden eerder ontworpen voor alleenstaand gebruik leidt tot een minder efficiënt multi-processor ontwerp. *(Deel I & III)*

3. Ingenuity and expediency are effective complements to analysis and synthesis. *(Chapter 6)*

4. De menselijke acceptatie van wijzigingen in de softwareinterface van verwerkingseenheden vereist een voorzichtige deconstructie van voormalige aannames en modellen; de bijbehorende aanpassingen in besturingssoftware zijn in vergelijking triviaal. *(Deel II)*

5. Amdahl's suggestion for a balanced design, i.e. a chip should provision a bit per second of external bandwidth for every instruction per second, should also apply to the bandwidth between any two individual hardware threads in multiprocessors. *(Chapters 9 & 13)*

6. Creativiteit en ijver zijn zelden samen aanwezig in één individu; het succesvolle uitbrengen van innovatie in informatica vereist dus een symbiose tussen verschillende persoonlijkheden. *(Dit proefschrift en het hele gebied van computerarchitectuur)*

7. The advent of general-purpose computers has altered the limitations of the human condition in ways both unforeseen and still poorly understood.

8. Het theoretische bestuderen van programmeertalen en computermodellen levert alleen kennis op over hoe mensen nadenken, niet over het gedrag van programma's.

9. Free and unencumbered information duplication is currently our sole means to avoid a digital dark age and leave a trace in history.

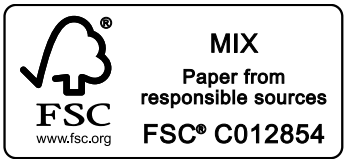
10. Videospelletjes en hun beleving door spelers zullen worden erkend door toekomstige historici als een baanbrekende nieuwe kunstvorm en een van de grootste prestaties van de mensheid.

On the realizability of hardware microthreading

Revisiting the general-purpose processor interface: consequences and challenges

This research was supported by the European Union
under grant numbers FP7-215216 (Apple-CORE) and FP7-248828 (ADVANCE).

Cover photograph by Éole Wind – `http://eole.me`



Typeset by LaTeX.
Printed and bound by Gildeprint Drukkerijen.

ISBN: 978-94-6108-320-3

On the realizability of hardware microthreading

Revisiting the general-purpose processor interface: consequences and challenges

Academisch Proefschrift

ter verkrijging van de graad van doctor
aan de Universiteit van Amsterdam
op gezag van de Rector Magnificus
prof. dr. D.C. van den Boom
ten overstaan van een door het college voor promoties ingestelde
commissie, in het openbaar te verdedigen in de Agnietenkapel
op woensdag 5 september 2012, te 10:00 uur

door

Raphael 'kena' Poss

geboren te Aix-en-Provence, Frankrijk.

Promotiecommissie:

Promotor:	Prof. dr.	C.R. Jesshope
Overige leden:	Prof. dr.	M. Brorsson
	Dr.	B. Chamberlain
	Dr.	C.U. Grelck
	Prof. dr.	P. Klint
	Prof. dr. ir.	C.T.A.M. de Laat
	Prof. dr.	S.-B. Scholz

Faculteit der Natuurwetenschappen, Wiskunde en Informatica

Even if there needs to be a shift at some point from the user's perspective—your goal is to make that shift as smooth as possible.

Daniel Prokesch

Contents

List of Figures

List of Tables

List of Side Notes

Listings

Summary

Ever since the turn of the century, fundamental energy and scalability issues have precluded further performance improvements in general-purpose uniprocessor chips. To "cut the Gordian knot," [RML+01] the industry has since shifted towards multiplying the number of processors on chip, creating increasing larger Chip Multi-Processors (CMPs) by processor counts, to take advantage of efficiency gains made possible by frequency scaling [RML+01, SA05]. Yet so far most general-purpose multi-core chips have been designed by grouping together processor cores that had been originally designed for single core, mostly single-threaded processor chips. After a decade of renewed interest in CMPs, the architecture community is barely coming to terms with the realization that traditional cores do not compose to create easily programmable general-purpose multi-core platforms.

Instead, the Computer Systems Architecture group at the University of Amsterdam proposes a general machine model and concurrency control protocol, relying on a novel individual core design with dedicated hardware support for concurrency management across multiple cores [PLY+12]. The key features of the design, described as "hardware microthreading," are asynchrony, i.e. the ability to tolerate operations with irregular and long latencies, fine-grained hardware multithreading, a scale-invariant programming model that captures clusters on chip of arbitrary sizes as single programming resources, and the transparent performance scaling of a single binary code across multiple cluster sizes. Its machine interface does not only provide native support for dataflow synchronisation and imperative parallel programming; it also departs from the traditional RISC vision by allowing programs to configure the number of hardware registers available by thread, replacing interrupts by thread creation as a means to signal asynchronous events, relying on a single virtual address space, and discouraging the use of main memory as an all-purpose synchronization device, preferring instead a specialized inter-core synchronization protocol.

The adoption of a different machine interface comes at the cost of a challenge: most operating software in use today to drive general-purpose hardware, namely operating systems, programming language run-time systems and code generators in compilers have been developed with the assumption that the underlying platform can be modelled by traditional RISC cores with individual MMUs grouped around a shared memory that can be used for synchronization. A port of existing operating software towards the proposed architecture is therefore non-trivial, because the machine interface *conceptually* diverges from established standards. In this dissertation, we investigate the impact of these conceptual changes on operating software.

We propose namely answers to the following questions:

1. Is it possible to program a chip with the proposed machine interface using an already accepted general-purpose programming language such as C?
2. What are the abstract features of the proposed machine interface that make it qualitatively different from contemporary general-purpose processor chips from the perspective of operating software?

The first question is relevant because the availability of existing programming languages is a prerequisite for adoption of a new general-purpose architecture. Moreover, support for C must be available before most higher-level software environments can be reused. For this question our answer is generally positive. By constructing a C compiler and parts of the accompanying language library, we demonstrate that programs following the platform independence guidelines set forth by the designers of C can be reused successfully on multiple instances of the proposed architecture. We also demonstrate how to extend the C language with new primitives that can drive the proposed hardware-based concurrency management protocol. However, we acknowledge that most programs also use system services and make assumptions about the topology and components of the underlying platform. We discuss why some of these assumptions cannot yet be adapted fully transparently to the proposed architecture and suggest a strategy for future work to do so.

The second question is relevant because its answer defines how to advertise the platform to system programmers, who constitute the early technology adopters with the strongest influence. For this question our answer considers separately the various peculiarities of the machine interface.

The proposed ISA provides native hardware support for thread management and scheduling and thus seems to conflict with the traditional role of operating software. Yet as we argue this support does not change existing machine abstractions qualitatively, because concurrency management was already captured in operating software behind APIs with semantics similar to those of the proposed hardware protocol. The machine interface provides configurable numbers of registers per hardware thread, which is a feature yet unheard of in other general-purpose processors. Yet as we show this feature can be hidden completely behind a C code generator, and can thus become invisible to operating system code or higher-level programming languages. The proposed chip topology promotes a single address space shared between processes, relying on capabilities [CLFL94] instead of address space virtualization for isolation, which diverges from the process model of general-purpose operating systems commonly in use today. Yet as we suggest the technology necessary to manage a single virtual address space is already available and widely used (for shared libraries and application "plug-ins") and this feature thus does not pose any new conceptual difficulty.

We found that the first conceptual innovation that warrants further theoretical investigation is the surrender of shared memory as the universal synchronization device for software. In traditional multi-core programming, implicit communication via coherent shared memory locations is routinely abused to provide locking, semaphores, barriers and all manners of time synchronization between concurrent activities. In the proposed architecture, such implicit communication is restricted and new basic programming constructs with fundamentally different semantics must be used instead. We formalize a subset of these semantics, then suggest how they can be used at a higher level in existing concurent programming languages. To summarize, despite their strong conceptual divergence, we found that these

new forms of synchronization are fully general and can theoretically be integrated in existing software, although more work will be needed to actually realize this integration.

The second conceptual innovation we found is the finiteness of concurrency resources. In most implementations of multi-threading on general-purpose platforms, threads and synchronization devices (e.g. mutex locks) are logical concepts instantiated by software, with the assumption that they can be virtualized at will using main memory as a backing store. In the proposed architecture, threads and synchronization devices are finite resources which cause execution deadlock when programs attempt to create more of them than are available in hardware. We argue that thread virtualization is not necessary with the advent of declarative concurrency in programming languages. Unfortunately, we cannot determine whether finiteness of synchronization devices is a net benefit towards further adoption of the architectural concepts. To summarize, we found that this innovation will require further analysis and possibly further refinements of the proposed architecture.

Beyond the scientific contribution towards the two questions outlined above, this book also contains a narrative about the design and implementation of a new processor chip in the midst of contemporary technology challenges. In particular, we highlight that a common trend in processor architecture research is to "solve" conceptual issues by abandoning support for some programming styles and software abstractions. We condemn this trend as being detrimental to true general-purpose computing, and we discuss throughout our dissertation how the requirement to preserve generality in processors impacts development, from the architecture up to applications via operating software.

Samenvatting

Sinds het begin van de eenentwintigste eeuw hebben energie- en schaalbaarheidsproblemen prestatieverbeteringen van individuele verwerkingseenheden zeer gehinderd [RML+01]. Om de spreekwoordelijke Gordiaanse knoop door te hakken heeft de industrie zich sedertdien gericht op het laten toenemen van het aantal verwerkingseenheden per geïntegreerd circuit. Dit heeft tot steeds grotere zgn. multi-processors geleid, waarmee aan (energie)efficiëntie gewonnen kan worden, door middel van het schalen van klokfrequenties [RML+01, SA05]. In het ontwerp van zulke multi-processors is het echter nog steeds gebruikelijk algemene verwerkingseenheden, ontworpen voor alleenstaand gebruik, samen te voegen tot een systeem. Een decennium na de herintrede van multi-processors is de computerarchitectuurgemeenschap nog altijd niet tot het inzicht gekomen dat multi-processors ontworpen door het samenstellen van traditionele verwerkingseenheden onvoldoende niet goed programmeerbar zijn voor algemeen gebruik.

De CSA-leerstoel aan de Universiteit van Amsterdam legt zich toe op het ontwikkelen van algemene machinemodellen en beheermodellen voor multiprogrammering, uitgaande van een nieuwe verwerkingseenheid naar eigen ontwerp, die hardwareondersteuning biedt voor de coördinatie van multiprogrammering over meerdere eenheden [PLY+12]. Een kernbegrip van deze nieuwe architectuur is "hardwaremicrodeeltaken." Dit begrip omvat asynchronie, d.i. om kunnen gaan met operaties met zeer uiteenlopende verwerkingstijden, fijnmazige hardwarematige deeltaken, een schaalonafhankelijk programmeermodel—welk clusters-op-een-chip van willekeurige omvang als enkelvoudige verwerkingsbronnen behandelt—en een in hoge mate transparante schaalbaarheid van de prestaties van binaire code over variërende clustergroottes. Buiten ingebouwde ondersteuning voor dataflowsynchronisatie en imperatieve parallelle programmeering wijkt deze verwerkingseenheid ook af van het RISC-paradigma op vier gebieden. Het laat programma's zelf bepalen hoeveel hardwarematige registers aan ieder deelproces worden toegewezen. Onderbrekingssignalen worden als primaire asynchrone signalering vervangen door primitieven voor het afsplitsen en samenvoegen van deelprocessen. De programmeur wordt een omvattende virtuele adresruimte aangeboden. Tevens wordt het gebruik van het werkgeheugen als primair synchronisatiemiddel sterk ontmoedigd—ter vervanging is een hiertoe toegespitst synchronisatieprotocol tussen verwerkingseenheden beschikbaar.

De introductie van een dergelijke nieuwe machineinterface heeft behoorlijk wat voeten in de aarde. Gangbare besturingssoftware (besturingssystemen, vertalers, looptijdomgevingen, enz.) veronderstelt veelal, dat de hardware waarop ze wordt uitgevoerd uitwisselbaar is met traditionele RISC eenheden met ieder een individuele MMUs, die allemaal toegang hebben tot hetzelfde werkgeheugen waarmee synchronisatie kan worden gerealiseerd. Dientengevolge kunnen bestaande besturingssystemen niet triviaal worden vertaald naar voornoemde nieuwe architectuur. Het probleem is dat de machineinterface op *conceptueel* niveau afwijkt van

gangbare standaarden. Dit proefschrift behandelt de implicaties van deze koerswijziging voor besturingssoftware.

Wij richten ons in dit proefschrift hoofdzakelijk op de volgende vragen:

1. Kunnen gangbare programmeertalen, zoals C, worden aangewend om dergelijke verwerkingseenheden te programmeren?
2. Wat zijn de kwalitatieve verschillen tussen de voorgestelde machineinterface enerzijds en dat van gangbare algemeen toepasbare processors anderzijds en wat zijn hiervan de implicaties voor besturingssoftware?

De eerste vraag vindt haar relevantie in het gegeven dat de aanvaarding van nieuwe architecturen voor algemeen gebruik veelal sterk afhangt van de overdraagbaarheid van bestaande programmeertalen. In het bijzonder moet C vertaalbaar zijn naar de nieuwe omgeving om menige softwareomgeving te kunnen hergebruiken. In grote lijnen kan deze vraag positief worden beantwoord. Met behulp van een hiertoe ontwikkelde C-vertaler en gedeeltelijk standaard C-bibliotheek kunnen we aantonen dat programma's uitvoerbaar zijn op verschillende instanties van de voorgestelde architectuur, mits deze zijn geschreven volgens de door de C-ontwerpers aangegeven richtlijnen voor platformafhankelijkheid. Tevens breiden we C uit met primitieven die de voorgestelde hardwarematige multiprogrammering aan kunnen sturen. Het behoeft echter vermelding, dat menig concreet programma verdergaande aannames doet over het systeem waarop het wordt uitgevoerd. We behandelen de pijnpunten van de overdraagbaarheid van deze aannames en doen suggesties voor toekomstig onderzoek te dezer zake.

De tweede vraag is relevant, aangezien het antwoord goeddeels bepalend is voor hoe de nieuwe omgeving ten beste aangeboden kan worden aan systeemprogrammeurs. Deze gemeenschap is in hoge mate bepalend voor de verdere aanvaarding van een nieuwe architectuur. De beantwoording van deze vraag gaat dieper in op de verschillende eigenaardigheden van de machineinterface.

De voorgestelde ISA biedt hardwareondersteuning voor het beheer en de planning van deelprocessen, wat ogenschijnlijk botst met de traditionele rol van besturingssoftware. Wij beweren echter, dat het niet (kwalitatief) strijdig is met bestaande machineabstracties, daar het beheer van multi-programmering in besturingssoftware zich normaliter achter APIs verschuilt met een vergelijkbare semantiek. Het machinemodel biedt controle over de toewijzing van het aantal registers aan deelprocessen, wat tot op heden ongehoord is gebleken in algemene verwerkingseenheden. Desalniettemin tonen wij aan dat deze eigenschap kan worden verhult door toepassing van een C-codegenerator, waardoor noch besturingssysteem, noch enige programmeertaal op een hoger abstractieniveau hier enige hinder van ondervindt. De voorgestelde chiptopologie biedt processen allemaal dezelfde adresruimte. Hier vindt toegangscontrole plaats op basis van geschiktheid [CLFL94], in plaats van virtualisatie van de adresruimte. Deze aanpak wijkt af van de procesmodellen van moderne, gangbare, algemene besturingssystemen. Evenwel wijzen we op de beschikbaarheid van gebruikelijke technologie voor het beheer van dergelijke onverdeelde adresruimten (zoals, bijvoorbeeld, voor gedeelde bibliotheken en dynamische programmauitbreidingen) die op vergelijkbare wijze kunnen worden toegepast, waardoor er geen conceptuele obstakels worden geboden.

De eerste conceptuele innovatie, waaruit verdere theoretische onderzoeksvragen voortkomen, is het loslaten van het idee om het gedeelde werkgeheugen als synchronisatiemechaniek

aan te wenden voor software. Traditionele multiprogrammeringsmethoden implementeren doorgaans abusievelijk met gedeelde variabelen wederzijdse uitsluiting, semaforen, synchronisatiebarrières en velerlei andere synchronisatiemechanismen tussen verschillende taken. De voorgestelde architectuur stelt paal en perk aan dit soort communicatie en biedt ter vervanging basis programmeerconcepten met fundamenteel andere semantiek. Een deel van deze semantiek wordt in dit proefschrift geformaliseerd en de toepassing van deze concepten in bestaande programmeertalen op hoger niveau wordt toegelicht. Samenvattend beweren wij dat, ondanks conceptuele afwijkingen, deze nieuwe synchronisatiemechanieken niets inboeten aan algemeenheid en theoretisch geïntegreerd kunnen worden met bestaande software, alhoewel voor dit laatste meer werk noodzakelijk is.

De tweede conceptuele innovatie die we presenteren is de eindigheid van de ondersteunende middelen voor multiprogrammering. Verreweg de meeste implementaties van algemene, multiprogrammeerbare systemen zien deeltaken en synchronisatiemechanieken (b.v. semaforen) als logische concepten die door software worden geïnstantieerd. Hierbij wordt aangenomen dat ze onbeperkt kunnen worden gevirtualiseerd in het werkgeheugen. In de voorgestelde architectuur zijn deze middelen eindig, waardoor programma's vastlopen wanneer ze zich meer dan in de hardware beschikbare middelen proberen toe te kennen. Wij bepleiten echter, dat met de opmars van declaratieve multiprogrammering in programmeertalen deze virtualisatiebehoefte teniet doet. Helaas kunnen we niet inschatten of de eindigheid van synchronisatiemechanieken de aanvaarding van deze architectuur in de vakgemeenschap in de weg zal staan. Samenvattend kunnen we stellen dat deze innovatie verder zal moeten worden onderzocht en dat de voorgestelde architectuur mogelijk zal moeten worden bijgestuurd.

Buiten de wetenschappelijke bijdragen omtrent de bovengenoemde vragen omvat dit proefschrift ook een relaas over het ontwerp en de implementatie van nieuwe verwerkingseenheden, te midden van hedendaagse technologische uitdagingen. We benadrukken in het bijzonder, dat recente onderzoeken naar architecturen van verwerkingseenheden meestal neigen naar het loslaten van het bieden van ondersteuning voor gevestigde programmeerstijlen en softwareabstracties om conceptuele problemen "op te lossen." Deze neiging wijzen wij af, als zijnde schadelijk voor algemeen computergebruik. Als rode draad door dit proefschrift loopt ons pleidooi voor het behoud van algemeenheid in verwerkingseenheden en de consequenties hiervan voor ontwikkeling, van architectuur, via besturingssysteem, tot aan toepassingen.

Dankwoord / Acknowledgements

In memoriam: Bertrand Russell, Dennis Ritchie & de kat

Zo'n onderneming van meerdere jaren, met zo weinig zekerheid op voorhand over de uiteindelijke resultaten, kan worden vergeleken met het opvoeden van een kind of met een zelf-ontdekkingsreis. Zo heb ik het wel gedeeltelijk ervaren; al was mijn project een zoektocht naar het onderliggende netwerk van kennissen, culturen, vaardigheden en methoden die aan het bouwen van een computersysteem ten grondslag ligt. Op zoek naar een waarheid die ik kon erkennen, heb ik dankbaar hulp aanvaard tijdens mijn oorlog tegen foute aannames, onvolledige argumenten, drogredenen en andere vormen van onwetendheid.

Deze ervaring durf ik te vergelijken met de cursus stijldansen waarvan ik vele zondagen gedurende mijn onderzoek genoot. Mijn meest betrouwbare danspartner is zonder twijfel Mike Lankamp geweest. Vier jaar lang hebben wij samen een genadeloze tango gedanst, altijd in strijd met elkaar, waarbij de rollen van leider en volger regelmatig wisselden, maar waarbij we altijd samen toe werkten naar een esthetisch elegant resultaat. Our training was faithfully guided by Chris Jesshope, who, beyond playing the background scientific tune to which we danced, also subtly gave the measure, provided us with first steps to follow, and trusted us to "make things happen" on our own. I will be forever grateful to both for this lasting learning experience.

While exercising, I have also learned much from working with my fellow dancers and elders: Irfan, Jian, Joe, Jony, Kostas, Michael and Qiang, you have been inspiring examples. Vooral Thomas en Michiel hebben mij vaak geholpen met de knepen van het vak en persoonlijke coaching; ik ben jullie hiervoor bijzonder dankbaar. Wij deelden ook de vloer met collega's die een andere stijl dansten. Al leek hun muziek op de onze, toch waren de pasjes net iets anders; door naar hen te kijken heb ik mijn eigen kunnen verbeteren. Daarom wil ik mijn respect uitspreken voor Andy, Clemens, Fangyong, Mark, Merijn, Peter, Roberta, Roeland, Roy, Toktam en Wei alsmede hen bedanken. Similarly, I cherished our joint learning with partners from other schools: Bodo, Carl, Dan, Dimitris, Frank, George, Jara, Leoš, Lukas, Martin, Raimund and Stephan, although you exercised a different swing on the same music, the experience would have been largely incomplete without you.

Naturally, I would like to also thank respectfully the members of the evaluation committee, namely professor Jesshope, dr. Chamberlain, professor Scholz, dr. Grelck, professor de Laat, professor Brorsson, and professor Klint, for their extensive comments and suggestions toward improving the earlier versions of this book.

Ondanks mijn liefde voor het dansen en mijn interesse voor de muziek van Chris, had ik het niet zo lang vol kunnen houden zonder de betrokkenheid en ondersteuning van een paar waardevolle en bijzondere personen. In de eerste plaats Rob Jelier, wie mij de weg van een promotie wees. Van Rob kreeg ik een vernieuwd geloof in de wetenschap, en ook

in mijzelf. Door zijn voorbeeld te volgen, heb ik mijn grondgedachten onderzocht, mijn dieptepunt van onzekerheid bereikt en ontdaan van alle externe motivaties. Toch ging ik door, gedreven door pure nieuwsgierigheid. Naast Rob zijn Mark Thompson en Simon Polstra trouwe vrienden geworden die me voortdurend aan herinnerden dat ik niet slechts een wetenschappelijk instrument ben, en dat mijn irrationele behoeften als mens ook aandacht verdienen. Want ondersteuning op het menselijke vlak had ik natuurlijk ook nodig: die heb ik niet alleen gekregen van Rob, Mark en Simon, maar ook van Marseda Duma, Sabien Jesse, Roberta Piscitelli, Arnold Pols and Sylvain Viollon, die er altijd voor me waren wanneer motivatie ontbrak. En toen de tijd kwam om na te deken over wat ik "erna" ging doen, heeft Merijn Verstraaten mij geholpen om een nieuwe betekenis aan mijn rol te geven; dankzij hem heb ik het afronden van het werk kunnen ervaren meer als een overgang dan een eindstreep. Zonder jullie was het werk en daarmee ook dit boek nooit afgekomen.

Verder is mijn werk ontzettend verrijkt door een paar onwaarschijnlijke ontmoetingen. Dankzij Marjolein op den Brouw heb ik begrepen dat wetenschappelijke onderzoek een zeer persoonlijke onderneming is, onafhankelijk van de voorkeuren van collega's en promotor(e)(s). Both Marjolein and Daniel Prokesch have taught me how inspiration and curiosity come from within and must often be carefully balanced with other life priorities. Het begrip dat onderzoek en onderwijs niet veel meer dan een sociaal spel zijn, met eigen politieke regels, leerde ik ten eerste van Philip Hölzenspies; van hem, en ook van Jan Kuper, leerde ik tegelijkertijd dat "het spel" leuk kan blijven, zolang de deelnemers ervan weten te genieten. Ik dank jullie beide voor de wijze les om vaker te focussen op wat voor mij leuk is.

Last but not least, I wholeheartedly dedicate a kind word to Stephan Herhut, Andrei Matei, Frank Penczek and Merijn Verstraaten. Despite, or perhaps thanks to, our contrasting personalities and quite distinct perspectives, you have provided a true meaning to the words "community of peers." With you I have simultaneously enjoyed the feeling of mutual interest, admiration, pride and respect, the excitement of working and learning together, and a healthy dosis of mutual evaluation and competition. I am looking forward to building our friendships further.

kena
Amsterdam, July 2012

Preface

Zonder verhaal zijn feiten sprakeloos.

Karel van der Toorn

When I joined the CSA group in September 2008, my supervisor Chris Jesshope tasked me thus: *"We made this novel chip called a Microgrid, and we want a C compiler to program it. Do it."*

This is most of what this book is about. You are now holding a report on four years worth of work towards compiling and running C programs on Microgrids, which are multicore, multithreaded processor chips. I am proud and content to report that a C compiler now exists for this architecture. The following pages will hopefully convince you that the corresponding software technology and example uses constitute an adequate answer to my supervisor's request.

Meanwhile, these four years have been intellectually instructive.

The first and most prominent discovery was that these Microgrid chips, that needed a compiler, lacked a specification when my work started. As I learned quickly, there are two meanings for the word "exist" in computer architecture. One definition, commonly used by the layman, suggests a working physical artifact. This definition, however, is quite uncommon in the research field: most chip architectures "exist" instead as *models* (blueprints) that can be studied analytically or via software simulations. The Microgrid architecture obviously did not exist in this first sense, as it was not expected anyway. My surprise, instead, was to realize that Microgrids only "existed" in the other sense in the *mind* of my colleagues. Their existence was vernacular, shapeshifting, and lacked committed documents that I could study and refer to. The second, most exciting discovery was that my colleagues had not invented *one* chip design but instead a *class* of chip designs, which could be instantiated over a range of design parameters. Therefore, to make a compiler, I first had to capture my colleague's minds on paper, then understand the architecture class so that a C compiler could target all its possible instances.

On this quest to gather such a *generic platform specification*, a prerequisite to implement a compiler, I made two further discoveries:

- previously written information about the new architecture was inconsistent, i.e. it expressed conflicting ideas;
- the unwritten intellectual model manipulated by my peers, while relatively consistent, was largely insufficient to implement a C compiler and accompanying software. For example, it missed the concept of external input and output.

In other words, the task initially assigned to me required that I first document the meta-architecture and then an instance thereof in sufficient detail. It also required that I complement the model with additional features required by the new software. While doing this, I had to accommodate the fact that some system features required to compile the C language are not technically part of a compiler nor the hardware architecture, but rather the "environment" provided by a software operating system. Therefore I had to also contribute some additional technology in that direction. All this happened, although as you would probably agree these steps are somewhat difficult to capture behind a simple "problem statement" such as found in most doctoral theses.

When I wrote the first version of this book, its narrative was similar to the steps outlined above and amounted to a straightforward technical report. When I initially submitted this report, my supervisor rejected it. He argued that the *scientific* community would not be interested in an engineering story; that they would rather like to understand which *new knowledge* I had created on my way. So I sat and mulled over the report. What did I learn during this work? What new knowledge did I create that was worth sharing?

At the foreground, surely there was some knowledge to share about the creation and exploitation of a C compiler for a new architecture. So I left that in, to be found in chapters 6, 8 to 11 and 13. Yet I found that sharing *only* this knowledge was oddly unsatisfying, because it depended on knowledge about said new architecture that was either unpublished or inconsistent, and somewhat incomplete. Therefore, I decided to *also* share the detailed knowledge I gathered from my colleagues about their Microgrid meta-model, complemented by details about the different concrete instances that I used in my work. And so I added that knowledge to the text, to be found in chapters 3 to 5. As a welcome side effect of this excercise, I was also able to argue towards a *renewed motivation* (chapter 2) for the architecture design, to complement the 15 years old argument [BJM96] that had so far served unchanged as justification for the line of work.

When this was done, I mulled again: although I had formulated new knowledge and made a scientific contribution to my field, I had not yet done justice to what I had found most important and fundamental in those four years. For there were two other aspects, which I dare call "knowledge" although my supervisor calls them "experience," that I had also isolated. The first is my understanding of the thrust of practitioners in my field: what *innovation* really means in computer architecture and why it is so crucially important to the future of computing. I share my views on this topic in chapters 1 and 16; this explains why you should care about my story, even though you may not be interested in Microgrids specifically. The second is my understanding of an obstacle shared between computer architects working on multi-core chip designs, that is the necessary disruption of unspoken knowledge.

Beyond the immediate need for more performance per watt, the role of a processor chip designer is to provide components to other humans to use in larger systems; as such, there is an iterated step of human-to-human communication required between component architects, system architects and software practitioners. Due to the largely technical nature of the devices considered, this communication is necessarily dense; to reduce verbosity, the people involved rely on unspoken, tacit knowledge constituted by their cultural background in computer science. *Innovation is difficult because of this reliance on tacit knowledge.* Any novel component will break some unspoken assumptions and invalidate some commonly accepted "truths." Innovation thus requires formulating a new cultural background about

hardware platforms, a step which most audiences—especially in the software world—are ill-trained to perform. In the case of Microgrids, my colleagues had already discovered that the mere prospect of additional performance is not enticing enough when its cost is a necessarily more complex mental model of the machine. What I have understood however, is that software audiences are willing to pay the price of more complex models if *they believe that they gain in flexibility*, i.e. future increases in productivity. There are two requirements to support this belief. The first is that the chip designer *does not gratuitously remove features* previously relied upon; the second is that the chip designer argues for the *generality* of any new features, to convince software audiences they can be used in ways not yet foreseen. As I found out, these two requirements were previously not very well addressed in the work around Microgrids. To my colleagues' defense, neither are they in other research projects in computer architecture. Nevertheless I believe they are still essential to the eventual fruition of innovative research in the field.

To advertise this experience, I thus decided to spend extra care to:

- detail the proposed architecture from the perspective of system software (chapters 3 and 4), highlight some of the most affected assumptions about the underlying machine model (chapter 7), and suggest how features commonly found in other platforms are also found in the proposed design (chapters 5 and 14);
- argue for generality throughout, starting with a renewed defense of general-purpose computing (chapter 1) and placing emphasis on the potential use of the platform by contemporary concurrent programming models (chapters 7, 9 and 12).

With this extra effort I want to communicate that the concepts around the advent of Microgrids are not merely the bread and butter of a small research group; that this research answers general requirements that deserve attention regardless of which specific architecture they are considered from. This is why chapters 3, 4 and 9 are longer than strictly required by a platform specification towards a C compiler, and why chapters 5, 7, 12 and 14 do not fit the purely technical line of thought of the other chapters. As I explain in section 1.7, this "meta-thesis" provides a second reading level, distinct from the technical contribution at the first level.

To keep a long story short, if you wish to restrict your reading to the joys of compiling C towards Microgrids, you can narrow down your focus to chapters 6, 8 to 11 and 13. The rest is "simply" *philosophy*.

Chapter 1

Introduction

> Ingenuity and imagination,
> rather than accurate thought, are
> the ordinary weapons of science.
>
> G.H. Hardy [Har11].

Contents

1.1 Epistemology: science is team work, so is innovation

The traditional purpose of the fundamental sciences is the acquisition of new knowledge pertaining to observed phenomena, in an attempt to describe "what is." In parallel to the discovery of new knowledge through scientific inquiry, philosophers, or theoreticians, derive ideas of "what could be." Via formalisms, they construct structures of thought to validate these ideas and derive iteratively new ideas from them.

We can focus for a moment on the human dynamics around these activities. On the one hand, the intellectual pleasure that internally motivates the human scientists is mostly to be found in the acquisition of knowledge and ideas. For natural scientists, the focus is on accuracy relative to the observed phenomena, whereas for philosophers the focus is on consistency. On the other hand, the external motivation for all fields of science, which materially sustains their activities, is the need of humans for either discovery or material benefits to their physical existence. From this position, the outcome of scientific inquiry and philosophical thought, namely knowledge and ideas, is not directly what human audiences are interested in. The "missing link" between scientific insight and its practical benefits is *innovation*, an *engineering* process in two steps.

The first step of innovation is *foundational engineering*: the creative, nearly artistic process where humans find a new way to assemble parts into a more complex artifact, following the inspiration and foreshadowing of their past study of knowledge and ideas, and guided by human-centered concerns. Foundational engineering, as an activity, consumes refined matter from the physical world and produces new more complex things, usually tools and machines, whose function and behavior are intricate, emergent composition of their parts. The novelty factor is key: the outcome must have characteristics yet unseen to qualify as foundational; merely reproducing the object would just qualify as manufacturing. The characteristic human factor in this foundational step is *creativity*, which corresponds to the serendipitously successful, mostly irrationally motivated selection of ideas, knowledge and material components in a way that only reveals itself as useful, and thus can only be justified, *a posteriori*.

The other step is *applicative engineering*, where humans assemble artifacts previously engineerd into complex systems that satisfy the needs of fellow humans. In contrast to foundational engineering, the characteristic human factor here is meticulousness in the realization and scrupulousness in recognizing and following an audience's expectations—if not fabricating them on the spot.

The entire system of activities around science is driven by a *demand for applications*: the need of mankind to improve its condition creates demand for man-made systems that solve its problems, which in turn creates demand for new sorts of devices and artifacts to construct these systems, which in turn creates demand for basic materials as input, on the one hand, and intellectual diversity and background in the form of knowledge and ideas. We illustrate this general view in fig. 1.1, and argue it is also valid for fundamental sciences in side note 1.1. The role of education, in turn, is to act as a glue, ensuring that the output of the various activities are duly and faithfully communicated to the interested parties.

Our first observation, which is perhaps the key motivation for the work presented in this dissertation, is that different humans who partake in these activities have different *preferences*. We do not expect any one person to participate in and be successful at all steps to improving the condition of mankind. The corollary is that for all processes to be successful, humans must *acknowledge their separate interests* and coordinate their work towards the common goals.

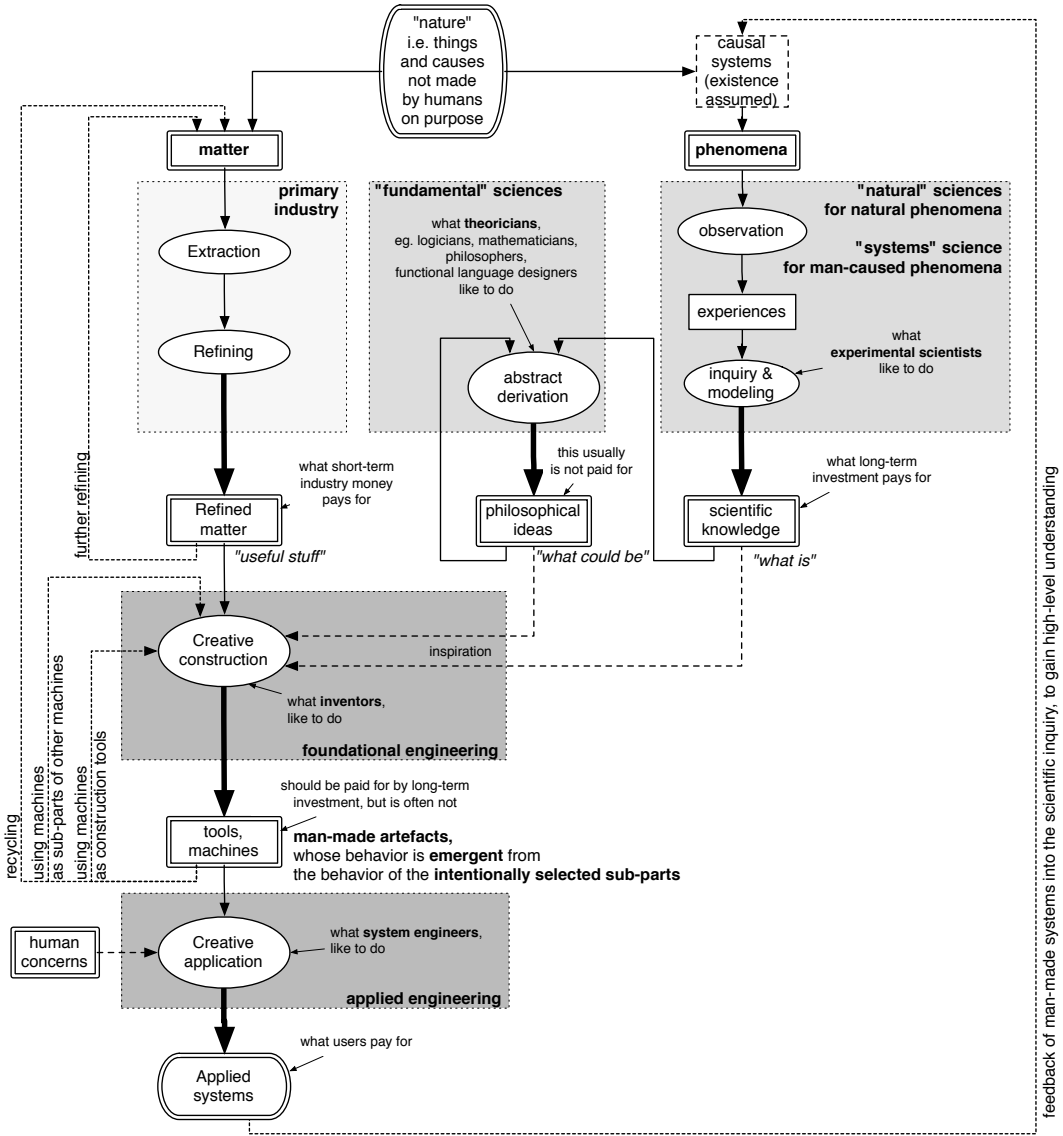


Figure 1.1: Activities related to science.

Side note 1.1: Fundamental sciences are also application-driven.

One may argue that the search for knowledge and understanding can be an end in itself, and deserve support (e.g. via funding) regardless of its applicability. However, we argue that *all* scientific activities are motivated by human needs for either discovery or comfort, and that it is human demand for their utilitarian value that fuels continued inquiry.
The need for discovery concerns the expansion of mankind, both over time, space and intellectually. Discovery has been historically more valued; it has fueled e.g. investigation into navigation and astronomy to conquer space, history and medicine to conquer time, mathematics, logic and linguistics to conquer the mind, theology, politicology and sociology to conquer the human group. Even astrophysics, quantum physics and cosmology can be seen as discovery means towards conquering the universe. All other fields of science exist to either improve the scientific process towards discovery, e.g. artificial intelligence or computational sciences, or to optimize the use of the already-conquered area by humans, i.e. increase comfort.
In either case, the outcome of the scientific activity must be *engineered* into useful objects before it becomes visible, even when the objects are just *knowledge vehicles* like books or lectures. Creating a book, for example, involves a foundational step in expressing the text by the author, then assembly of the text into a concrete print by the publisher. Even with "fundamental sciences" which claim to produce only abstract knowledge, the value of the scientific activity can only be ascertained once the acquired knowledge is actually communicated. We can thus say that it is the demand for knowledge vehicles, expected as physical artifacts, that fuels the fundamental sciences.

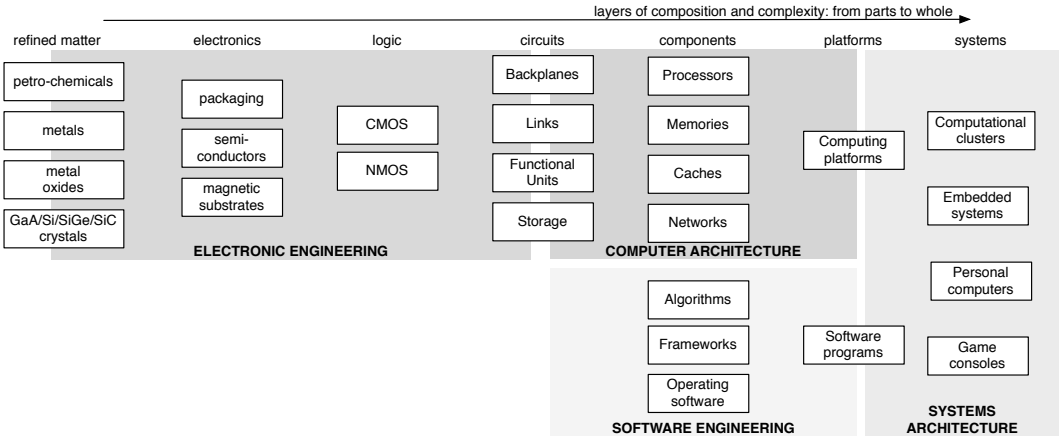


Figure 1.2: Composition of parts to entire computing systems.

1.2 Computer science, engineering, and its architecture sub-field

The term "computer science" broadly encompasses all activities around the design, manufacturing, application and analysis of computing systems. Computer systems engineering is its subset of activities related to the design of the computing artifacts, i.e. the design of tools that can be subsequently applied by the IT industry to mankind's interests. We illustrate the further sub-fields of computer engineering in fig. 1.2, namely: electrical engineering, software engineering, computer architecture and systems architecture. As highlighted above, the foundational part of computer systems engineering is inspired by, and based on knowledge and ideas from natural sciences, such as biology and physics, and philosophy, such as the theoretical work of George Boole, Alan Turing, Alonzo Church and Claude Shannon; its outcome, i.e. computing systems, is then instantiated in computing applications, such as home appliances, cars or medical equipment.

In contrast to some fields of technology that can distribute the various activities to separate groups of people, creating a divide of social roles between "scientists," assumed to be out of touch with the needs of daily life, and "engineers," assumed to be out of touch with fundamental theory, computer science is peculiar in that it requires all its practitioners to be both skilled engineers and competent fundamental and experimental scientists. This is a characteristic shared by all fields where the produced systems are complex assemblies of parts themselves complex, deployed in complex situations; avionics, high-speed trains and spatial equipment are other such fields. The reason is that for the composition of complex pieces into a complex system to be tractable by human practitioners, first a model of the parts' behavior must be devised, to simplify the mental image manipulated in the creative process in the next step. For example, when computer architects assemble caches and processors into a multi-core microprocessor chip, they do not keep track of the individual composition of the components in terms of p-type and n-type transistors; instead they manipulate an idealized model of their behavior derived from previous implementations by the experimental and analytic scientific inquiry. This abstraction process happens at all levels of composition, from the grouping of silicon crystals into semiconductors to the grouping of individual processing units into networks of computing clusters.

Historically, computer science was first an isolated field whose end applications were restricted to defense and high-profile business administration. States (for defense) and large companies (for business) could fund single organizations with large research facilities to design their systems. When computers became useful to other applications with more democratic audiences, the audiences could no longer directly fund single, large organizations. The design of computing tools became distributed across distinct organizations carrying the innovation process at different levels, and interacting together to combine systems. A key enabling factor for this division of concerns was the invention of the general-purpose computer, which allowed to separate the designers of hardware from the designers of software into distinct communities.

1.2.1 The wonder and promise of general-purpose computers

In the middle of the 20th century, something exceptional in the history of mankind happened: a *universal tool* was invented: the general-purpose computer. For the first time, any human could potentially become an inventor and solve arbitrarily complex problems using pictures of their thought processes projected as information patterns, i.e. programs, without the cost of manufacturing new physical artifacts.

Although Charles Babbage can be acknowledged as the forefather of this artifact [Hal70], the full scope of its generality was only understood by Turing at the start of the 20th century. Indeed, our current general-purpose computers approximating Turing's abstract machine are one of *only two ways* that we currently know to make a computer that *can compute most of what a human may want to compute*; the other way being queue machines, invented later [FE81]. Furthermore, when the computer is connected to input/output interfaces, it becomes *interactive* and can convert to and from phenomena of the physical world. Conversely, there is no known other way to assemble artifacts, other than those that can be described by interactive Turing and queue machines, which can compute most of what humans can think of in a way that can influence, or can be influenced by, physical phenomena.

Obviously one does not need to invoke Alan Turing nor his thoughts when building a device that accomplishes a specific task. Census count machines reading punched cards were built and used successfully long before Turing was born [Hol89, Ran82]. The reason

why Turing's contribution was remarkable is that it created *theoretical confidence* that a general-purpose hardware platform could be successfully reused, after it is fabricated, to solve problems not defined *yet*, and thus guaranteeing perpetual employment for software creators.

This confidence stems from the formal thesis of Alonzo Church and Alan Turing on computability—although it really comes from collective work with Post, Kleene and Gödel, cf. [Dav65] for the full story—which establishes that all expressible formulas of arithmetic, which by definition are all possible computations that humans can phrase intelligibly ever[1], can be computed by either an abstract Turing machine, Church's λ-calculus or partial recursive mathematical functions[2]. Moreover, when the machine is *universal*, the function it computes can become a run-time input, i.e. *software*, while preserving the full generality of the model. Because of this, a hardware platform that resembles a universal Turing machine gives us confidence that it can be reused in the future by new software to solve problems that have not been formulated yet. Since the only conceptual difference between a universal Turing machine and a concrete implementation is the finiteness of storage capacity (vs. the infinite length of Turing's tape), it is possible to approximate the abstract machine increasingly accurately by simply adding more addresses to the storage, which seems to be technically tractable for the foreseeable future.

This is the crux of what *general-purpose* computing is about: design and build a hardware platform now, with reasonable confidence founded in logic that they can be used to solve future problems in software.

Since then, other formalisms distinct from Turing machines and λ-calculus have been developed and subsequently proven to be *Turing complete*, that is, at least as powerful as Turing machines. The models that have received most attention are queue machines (mentioned above), graph reduction machines able to reduce terms of λ-calculus [CGMN80, PJCSH87], register-based variations on the Turing machine [vEB90], the π-calculus [Mil90], and specific initial states of cellular automata [Cha02, Coo04]. Any of these models can be made universal, i.e. programmable via software, by modeling a single-function machine whose behavior is to read a program from an input device and then evaluate it. They can furthermore be made interactive, i.e. connected to the physical world, by introducing basic actions in their evaluation rules that effect external interactions. However, the only computing artifacts built so far have been either Turing-like (register machines) or queue-like (stack machines). All implementations of other formally as powerful models have only existed as simulations of their semantics in programs running on hardware devices that can be primarily described as register or stack machines. It is the Turing-completeness of these specific two models that guarantees the *future utility* of the constructed artifacts.

1.3 General-purpose computers are the stem cells of computing

Sometime between 1992 and 1996, CALC was written. CALC was a graphing program: the user would interactively enter on the keyboard the definition of a function and the coordinates of a view window, and the program would plot the function, one point per

[1] NB: computations (computable functions) are a subset of all functions that can be phrased by humans. In particular there exist intelligibly phraseable non-computable functions, such as the busy beaver function [Rad62].

[2] It further establishes that neither of these formalisms can compute anything *more*, that is, everything we can compute using either can be described by an intelligible, valid formula of arithmetic; but this point is irrelevant here. See [Hof99] for an accessible review of the theoretical and practical consequences.

column of the graphical display. As the story goes, CALC was written in BASIC over the course of several months; a few months afterwards, the only extant copy of CALC was lost.

We resurrect the memory of CALC here to highlight the role of general-purpose computing. Indeed, CALC would allow the user to enter any function definition that was valid in BASIC. The syntax allowed integer and floating point arithmetic, grouping parentheses, operator precedence, and uses of any built-in functions. It would then plot that function interactively, i.e. without having to stop and re-run the program. In other words, the program would understand a phrase expressed in a human language, that of mathematics, and act upon it automatically. Yet, implementing that feature was trivial: CALC would simply write the text of the user-supplied expression into a file, and load back the file into the BASIC interpreter as an additional program fragment[3].

To understand how this is relevant here, one needs to consider this anecdote as a parable. What happened really is that an uneducated person was empowered to *create* by a programming environment which was, through its naive simplicity and despite its flaws, intendedly devoid of any specific purpose. A simple general feature, namely the ability to read a user-defined program text from input and evaluate it, was key to overcoming the most complex theoretical aspect of the task at hand. This parable illustrates that general-purpose computing platforms are, *like the stem cells of living organisms*, key to the perpetuation of computer engineering. They empower practitioners, both amateur and seasoned, to express their creativity past the bounds suggested by current applications and uses, and solve new problems in revolutionary ways.

There are two reasons why this generality is strongly desirable. The first reason is that innovation and major advances in the field are a creative process by humans for humans, as highlighted above. Creativity in humans usually occurs only in unbounded conceptual frameworks and playgrounds. Therefore, computing science, as a field, will need flexible and generic platforms for new developments and innovation. These platforms might be isolated, conceptually or physically, from the products available to the general public, but even when so pressured they will continue to exist as an essential and durable niche market for computer science practitioners themselves.

The second reason is that all current trends converge towards the second era of separated computing[4], with visible and much-awaited benefits in terms of energy and cost management.

The visible tip of this iceberg, on the network side, is perhaps the ongoing rise of social networks and online sharing platforms. But even in corporate environments, more and more responsibility, in particular regarding the safeguarding and consolidation of data, is pushed away from workstations to networked locations and accessed remotely. This setup principally enables sharing the infrastructure costs (security, cooling, storage, failure management) for the compute-intensive parts of networked applications. It reduces synchronization and communication latencies in large applications by literally increasing *locality*, namely by grouping the communication-intensive parts into a close geographical location. Through careful over-subscription of shared computers, it also distributes the energy investment more equally across heterogeneous applications. This setup is technically usable nowadays, as opposed to the last part of the previous century when the client-server model somewhat waned, essentially because of lower latencies in networks (cf. side note 1.2).

Meanwhile, and perhaps paradoxically, the devices at the human-computer interface become increasingly powerful. Current low-end gaming devices already offer full virtual immer-

[3]Using the `CHAIN MERGE` statement.

[4]The words "separated computing" include both the asymmetric client-server model and distributed applications where the overall behavior emerges from equal contributors.

sion through rich auto-stereoscopic images [JMW+03, Tab10, Lea10]. Reality-augmenting glasses with on-demand, real-time streaming of contextual data are on the verge of becoming mainstream [HBT07, Ber09]. All futuristic visions of human-centric computing include pervasive and seamless human-computer interaction with incredible (by today's standards) amounts of signal processing.

To maintain control on power usage and locality, most of the signal processing will need to be physically performed at the site of perception. What we currently call high-performance computing equipment will find its way to the wearable miniature sensors of our future selves. However, for no less obvious reasons, the processed data will flow between the individual and the collective self, through distributed networked applications, because only there can the sense-data receive the meaning necessary to its processing[5].

Without speculating further on the nature of these dual computing systems made of intelligent sensors and networked applications, it seems reasonable to assume they will be based on hardware components responsible for transforming information. These future systems may bear little resemblance to our current technology; yet, regardless of their exact nature, one of their characteristics seems inevitable: *adaptability*.

Adaptability is the feature that will support technological evolution under the selective pressure of market effects. Indeed, unless disaster strikes and totalitarian regimes become the norm, free exchange of ideas and objects will force a dynamic, fast-paced adaptation of technology to evolving human interests. Even assuming a stabilization of human demographics, the increasing access to technology and networks will cause the market for computing systems to become even more segmented than today, with entire verticals[6] rising and falling faster than education systems. Combined with the fact that the knowledge required to comprehend and maintain systems will be increasingly dense, and thus decreasingly accessible, there will not be enough manpower to design and implement entire new systems to cater for new verticals. Since there is not yet any confidence that new designs can be reached via autonomous artificial intelligence, we should assume instead that guided adaptation of existing concepts to new uses and new requirements *by humans* will be the norm.

Evolutionary theory suggests that adaptation works best if the system keeps a healthy reserve of repurposable stem cells. It seems conceptually difficult to re-purpose the programmable controller for a washing machine into a car navigation system; whereas the computer scientist today clearly sees a specialization path from a general-purpose computer to both devices. Actually, specialization of computing elements, like cell differentiation in organisms, is an unavoidable phenomenon required to support the increased complexity of their applications. However efficient specialization is a *repeating* phenomenon, with each generation stemming from non-specialized components instead of previous generations of specialized systems. This applies to both hardware design and software design.

In the light of this perspective, one could possibly accept the doom of commodity, one-size-fits-all "all-purpose" computer designs. Individual devices that would truly satisfy *any* computing need in a practical or economical way have never really existed. Besides, the immediate human-computer interface is best served by specialized devices. However, general-purpose *specializable* computing systems must continue to exist, at least for those humans who, through their creativity and inventiveness, will be responsible for future innovation.

[5] This vision of *networked computing* was inspired by professor Zhiwei Xu, from the Chinese Academy of Science.

[6] *vertical*: an entire computing market addressing a specific segment, from hardware manufacturing to online services including system and software development. Two archetypal verticals are mobile phones and car navigation systems.

Side note 1.2: The second rise of separated computing.

Historians of our field have highlighted that client-server computing has been prevalent before, but then disappeared. This warrants an aside.
The key observation is that the client-server model "makes sense" economically and practically for the reasons mentioned in section 1.3. These merits have never disappeared; they have been merely shadowed by the ever-increasing computing speed of autonomous, commodity general-purpose computers ("desktops") in the last period of the 20th century relative to the network latencies.
Indeed, the centralized structure of the first networks, combined with the characteristics of network links at that time, was a serious obstacle to their increased prevalence. To match the increasing expectations of computer users, applications had to become more interactive and more responsive. More than the load on computational performance on servers, the subjectively unacceptable response time and jitter of communication links was the first motivation towards increasingly powerful terminals and locally interactive applications. Previous work on human-computer interaction [Mil68, CMN83, CRM91] suggests that 100ms is the maximum acceptable delay between a keystroke and the screen response, and [Shn84] suggests that higher variability of response times incur lowered user performance. These findings have served as references for user interface design ever since [Nie94, Chap. 5]. In contrast, the high latency and contention of terminal channels to shared computing facilities in the period 1970-1990 was failing users on both counts.
The market for personal computing grew initially faster than communication networks, further strengthening the trend toward autonomous computation devices, incarnated in personal computers. The low-latency, high throughput, globally available distributed network that was needed to support the growth of a computing market based on the client-server model only appeared much later—and unfortunately long after audiences had gotten used to the merits and flaws of their autonomous, power-hungry, inefficient commodity general-purpose computers. In contrast, the reason why the Internet became what it is today was a combination of economic and social factors outside of the realm of computer science: the global need for more connectivity, more trade, information exchange, etc. The epic conflation of telephone and data networks is fueling, to this day, a fierce competition that produces as a side effect an increasingly large, robust and wider network.
Now that the global network is available, we should find it remarkable that all online actors, from individual news broadcasters to scientific computing centers, know acutely the huge potential of such a strong connectivity. It is just natural that we use this opportunity to again tap into the merits of separated computing, which we had left aside for a few decades.

Side note 1.3: Embedded/specialized vs. general-purpose computers.

A defense of general-purpose computing would be incomplete without a reference to embedded systems. The increasing pervasiveness of embedded systems is often cited as a tendency towards the disappearance of general-purpose computing. However, this is likely a fallacy.
The characteristic of embedded systems that is relevant here is their invisibility: while they are necessary to technological evolution, and their function is expected by the users of the devices where they are embedded, these systems are not shown. In fact, embedded systems do not have an existence as computers in the eye of their users, which therefore do not expect from them "abstract" features such as re-programmability and re-configurability. The large and growing embedded market is one of specialized function, reliability, sometime performance, and foremost dedication to well-defined tasks that allow economies of scale in manufacturing. Embedded devices are essentially engineered for specific purposes at the expense of programmability and configurability, which typically matter less to their audience.
When considering their manufacturing and sales figures, and the massive body of knowledge developed to support their field, embedded systems dominate the field of computer science by sheer numeric superiority. It is this pervasiveness that seems to shadow general-purpose computing and suggests its doom from the perspective of practitioners in the field.
However using the growth of embedded systems as an argument against general-purpose computers amounts to ignoring the "elephant in the room": the creation of embedded devices would not be possible without development computers, those very computers used by embedded system engineers: developers, testers, quality controllers, etc. By the initial argument above, most of the advances in embedded system design would not be possible without *stem computers* that let the designer imagine radically new devices without preconceptions: either their own development machines, or non-specialized, freely composable template components with no predetermined function.

Side note 1.4: Example functions that can be made primitive recursive.

Multiplying two matrices; filtering noise out of audio; compiling assembly code to machine code; computing a minimum flow in an acyclic graph; recognizing object boundaries in an image; plotting a graph for a fixed arithmetic function; printing a JPEG image; looking up a value in a dictionary; simulating another computer which can compute only primitive recursive functions.

Side note 1.5: Example functions that are not primitive recursive.

Sorting; recognizing a tune from an audio sample; compiling a C^{++} program to machine code; computing a shortest path in a cyclic graph; recognizing objects in an image; plotting a graph for an arithmetic function entered interactively; typesetting a TEX document; running an SQL query on a relational database; simulating another computer which can compute partial recursive functions.

1.4 Losing sight of generality: the risk of throwing out the baby with the bathwater

Half a century after having found a true wonder, we run the risk of losing it: under the pressure of capitalism, optimization and efficiency, the generality of computers has been endangered since the turn of the 21st century.

The prevailing thought in the personal computing industry at the time of this writing is that there are no uses of computing systems by the general public that are *freely programmable by the user*[7]. A common argument in favor of such an iWorld, where form is actualized by branded functions, and where each fixed-function iApplication is blessed individually by arbitrary and opaque corporate review processes—or euthanized before birth if its market value is not ascertained upon inception—is the growing enthusiasm for iThis and iThat devices initiated with the new millennium.

The forces driving this evolution are further discussed in [Zit08]: the free market and deregulation of large companies has created incentives to innovate behind closed doors and create instead devices that capture their users into consumption cycles. Capital gains discourage reuse and extension of existing products in favor of the forced acquisition of new products for every new application. The associated risk is an *opportunity loss*: individuals with the skills to innovate in software become increasingly limited in their ability to carry out innovation until they are trained privately by corporate technology providers.

Even in technical circles, where generality is traditionally respected, the need for increased computing power and efficiency creates pressure to reduce generality. This effect has existed throughout the history of computing; for example, the author of [Day11] reminds us how Dijkstra's suggestion to introduce recursion in programming languages was met with resistance from audiences focused on the short-term performance gains from non-recursive language semantics. Yet, for reasons we will revisit below in section 1.4.1, the computer engineering field has come lately under tremendous pressure to answer requests for additional performance, and the temptation is great to specialize devices to match this demand.

We can recognize this situation in the recent enthusiasm for "accelerator" boards, which re-purpose graphics-oriented co-processors (usually texture shaders) towards other types of computations. The marketing message is that these processors are "general-purpose": they seem to address application needs of both scientific and consumer audiences alike. However this message is misleading: full generality requires interaction, arbitrarily large

[7] Including, but not limited to, the ability to share freely the outcome of a programming activity with peers.

random-access storage and either arbitrary numbers of conditional branches or arbitrarily deep data-dependent recursions, *per individual thread*, which these devices usually do not support[8]. Instead, they are just sufficient to compute any *primitive recursive function* of arithmetic[9]. Since most of what humans need to compute can be described by primitive recursive functions (cf. side notes 1.4 and 1.5), this type of computer seems already quite useful; however, by the argument of section 1.2.1, these devices fundamentally limit our ability to solve future problems.

1.4.1 The current pressure to innovate

Processors and memories are two unavoidable sub-parts of any computing system, as they are at the heart of the ability of the system to compute (cf. section 1.2.1). Until the turn of the 21st century, system engineers using these components as building blocks could assume ever-increasing performance gains, by just substituting any of these components by the next generation of the same. Then they ran into two obstacles. One was the *memory wall* [WM95], i.e. the increasing divergence between the access time to memory and the execution time of single instructions. The second is the *sequential performance wall* [AHKB00, RML+01], i.e. the increasing divergence in single processors between performance gains by architectural optimizations and the power-area cost of these optimizations. A third potential wall is now visible on the horizon: the end of Moore's law [Kis02, ZCHB03, TP06], or more precisely a potential limit on the maximum number of silicon CMOS-based transistors per unit of area on chip. These obstacles are the stretch marks of a speculation bubble ready to burst. Unless solutions are devised within a decade, the economy of the IT industry of applications, currently based on an expectation of future cheap performance increases in computing systems, may need a serious, globally disruptive reform.

1.4.2 Where and how to innovate

The discussion so far opens space for innovation in different directions. A first possible direction is to invent a new implementable universal computing model that offers at least as much expressivity as Turing's model, with more computing power and scaling opportunities than what all current devices offer. Quantum or biological computers may be candidates, although their realizability at the scale of current human needs, and their universality, are not yet ascertained. Another direction is to find new smaller and faster building blocks to make universal and interactive Turing-complete computers, for example using light-based logic instead of electronics [SLJ+04], however these new technologies may not be available before the limits of silicon scaling are reached. In the medium term (a decade), innovation seems tractable at two levels. Software ecosystems could embark on an intensive quest for simplification towards fighting Wirth's law[10] and gaining efficiency. However, the odds that this will happen are at best unclear [Ken08, MSS10, XMA+10]. Or, *computer architects* could find new ways to arrange CMOS logic into different, more efficient combinations of processors and memories, possibly exploiting parallelism and scalable throughput. This direction, initially suggested in [RML+01], seems especially tractable given the growing availability of concurrency in software, a topic we revisit in chapter 2.

[8] We support this argument for an example state-of-the-art accelerator chip in chapter 12.

[9] To simplify, a primitive recursive function is a computation where the number of steps in any repetition is known before the repetition starts; for details see [Pét34].

[10] "*Software is getting slower more rapidly than hardware becomes faster.*" [Wir95]

1.4.3 Politics and openness in innovation

Assuming computer architects will be responsible for principal innovations in computer engineering in the next decade, and assuming generality in the produced systems is a desirable feature for software audiences, there exist two principal questions:

- the *inner question*, from the individual architect's perspective, is one of substance: *what* will be the good ideas and new component assemblies? What will they look like?
- the *outer question*, from the perspective of the computing science community, is one of logistics: *how* to ensure that any innovation, if it occurs, will be brought to realization and support the continued growth of computing ecosystems?

Answering the inner question is fully under the responsibility of the architects, and bounded only by their creativity. The outer question, however, is not under their control. As we identified in section 1.4, market effects combine the overall need for new technology with an incentive for corporations to keep their solutions specialized and opaque. Levers to market dominance and corporate power struggles, in particular increasing uses of the vendor lock in and planned obsolescence effects, create the risk that any coming innovation in computer architecture will be realized behind closed doors, and will only be featured in products lacking the desired generality.

The computer architecture community is responsible for safeguarding against this appropriation of upcoming innovations by corporate interests. The methodology to guarantee the generality of solutions and their accessibility to large audiences is obviously *transparency*, that is documenting publicly not only the outcome of the innovative processes but also *which creative steps were undertaken* and their background knowledge and experiences. This is necessary so that other parties, in particular younger generations of computer engineers, can steadily join the innovation enterprise via imitation, reproduction and extension of past discoveries.

1.5 The Achilles' heel of designers: HIMCYFIO

On the road to innovation, computer architects face a pitfall which we call "*Here Is My Component, You Figure It Out (HIMCYFIO).*"

The circumstance for HIMCYFIO concerns visionary computer architects who address a known or future technology obstacle while avoiding complexity, i.e. who deviate from established routes while preserving conceptual simplicity. The pitfall occurs when the inventor proposes a new component design, in isolation, with the unfounded confidence that its purpose and utility are both self-explanatory and desirable, and thus that it will be "necessarily" successful upon completion. The assumption is that industry will "eventually catch up" and integrate the component in larger computer systems, because the need seems self-explanatory (cf. section 1.4.1) and the integration self-evident (seems simple to the designer himself). This assumption is fallacious because peer inventors and potential users, especially the software community, will both find the invention foreign, i.e. difficult to understand simply because it differs from the common ground, and non-trivially applicable, because its proper use in larger computer systems is not yet determined.

To summarize, HIMCYFIO occurs because hardware architects mistakenly consider that the novelty and simplicity of a new approach is sufficient to relieve them of the burden of holding their audience's hand in appreciating their work. In even shorter words, HIMCYFIO occurs when architects answer their inner question while avoiding or answering incorrectly

the outer question (cf. section 1.4.3). When HIMCYFIO occurs, potentially interesting component designs run the risk of never being integrated in a working computer. Large research investments, including potential great ideas, may be lost; personal ambitions may be crushed. We could find several example victims of HIMCYFIO:

- the Manchester dataflow machine [GWG80] proposed a relatively simple processor design using dataflow principles, with the early vision that the latency of all external communication, even fine-grained memory accesses, should be tolerated by selecting independent instructions in hardware. However its exclusive dedication to SISAL [MSA+83] forced its designers into a niche community where they stayed until the termination of the project in 1995;
- the designers of the Monsoon architecture [PC90] had a similar combination of fresh applications of dataflow scheduling and practical implementation plans for a processor. However they restricted the use of their design to the Id language [ADNP88], while suggesting that I/O and program control would be handled on a separate type of processor running the (by-then-already-standard) Unix operating system. This forced users to work with two programming models, and the resulting conceptual conceptual complexity proved fatal to the project;
- a more recent and high-profile example can be found with Transmeta's Crusoe architecture [Kla00]. On the one hand, the combination of a relatively simple Very Large Instruction Word (VLIW) design with Code Morphing [DGB+03] was a revolutionary way to expose a high-ILP, low-power pipeline to an audience locked into Intel cores. On the other hand, the final product's packaging was ill-designed: despite the presence of an on-die DDR memory controller (for the instruction cache), external memory was supported only via a slower SDRAM interface; and I/O was restricted to the legacy "southbridge" system interface. These integration choices rendered the overall chip uncompetitive for data-intensive or graphics-intensive applications. While Transmeta's goal was to guarantee thermal budget envelopes, and the chip was retrospectively competitive in this regard, this priority was not appropriately communicated to the company's audience.

These examples illustrate that to avoid this pitfall, architects should design new components in the context of entire systems and their applications:

- when designing for new applications, they must work together with system and compiler programmers so that this first level community obtains an early understanding of the architecture's benefits;
- when designing for existing applications, they must acknowledge the assumptions made in the existing programs and provide designs that integrate existing usages transparently; in particular, they must acknowledge common expectations about performance without requiring radically new machine models.

The reason why recognizing and averting HIMCYFIO systematically is important is that this is the only way to make generative architecture research an attractive field for newcomers.

1.6 Case study: hardware microthreading

In an effort to innovate in computer architecture while preserving generality, as identified in section 1.4, a research project coordinated from the University of Amsterdam is exploring new ways to assemble microprocessors from logic. After observing that a large amount of untapped concurrency is already present in software applications, the researchers in this group propose to design processor chips that cater primarily to throughput scalability and execution efficiency, possibly at the expense of the performance of single instruction streams. The proposed chip design combines short Reduced Instruction Set Computer (RISC) pipelines with dynamic dataflow scheduling [NA89] and Hardware Multi-Threading (HMT) on individual cores for latency tolerance and energy efficiency, and implement flexible hardware support for thread and task management across different cores for throughput scalability. We shall call this approach "hardware microthreading," as opposed to simply "microthreading" used in prior work from this research team, to distinguish it from the microthreads managed in software in IBM's Cell Broadband Engine [AAR08].

The abstract definition of hardware microthreading, together with guidelines on how to design architectures around it, establishes *principles* of architecture design. These should in turn be applicable to a diversity of application domains, from embedded systems to high-performance supercomputers. However, before this happens the principles must be illustrated at a small scale, into an artifact suitable to convince external observers, as highlighted in section 1.5.

To achieve this, the research group also attempts to define an entire general-purpose chip architecture which embodies hardware microthreading with simple RISC cores and a custom Network-on-Chip (NoC). In the preliminary phase, given the complexity of modern silicon, this research was carried out via detailed simulations of individual components based on their potential behavior on silicon, using artificial microbenchmarks. The initial results were encouraging [BHJ06a, BHJ06b, BGJL08], and motivated the enterprise of a larger project to carry out a more extensive realization:

- at the hardware level, design and implement a prototype single-core implementation of the new architecture on an Field-Programmable Gate Array (FPGA); and simultaneously implement a software emulation of a full multi-core system using the new processor design;
- at the software level, design and implement new operating systems, language tool chains, and a representative set of benchmarks to evaluate the new hardware architecture, both on single core (FPGA prototype) and multi-core systems (emulation environment).

This is the project where this book originated.

1.7 Overview

Our dissertation reflects on the interaction between foundational engineering activities around hardware microthreading. It provides two reading levels.

At the first level, a technical report provides possible concrete answers to the outer question around hardware microthreading. This report recollects the substance of the invention in part I, then describes in part II how we extended the base architecture concepts into a

Symbol	Description
○	Exposition of previous answers to the inner question.
□	Argument or contribution towards answering the outer question.
■	Contribution towards answering the inner question, required while answering the outer question.
▷	Acknowledged opportunity for further work on the outer question.
▶	Required further work on the inner question, identified while answering the outer question.

Table 1.1: Symbols used to mark technical contributions throughout our dissertation.

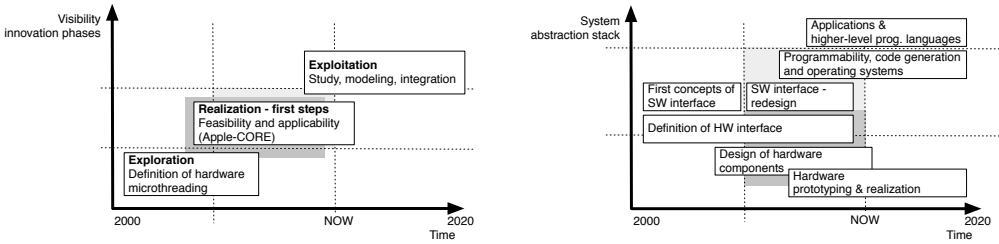
general-purpose platform, and finally summarizes key evaluation activities around this platform in part III. During this research, we also made related minor contributions to the substance of the invention. We classify our findings about the inner and outer questions around hardware microthreading throughout the text using the symbols from table 1.1; a summary is given in table 1.2. This technical contribution touches multiple system-wide issues, from code generation to applications through the operating system stack; as such, it *connects architecture and system research* [MBRS11]. The scope of the work at this reading level is illustrated as the light gray area in fig. 1.3: this dissertation complements [Lan07, Lan1x], which mostly describe the answers to the inner question (dark gray in the figure); it also does not consider how the innovation will eventually be used in specific applications.

The main contributions are to be found at the second reading level. Our choice of hardware microthreading is really a case study for the dialogue between the innovator and their community of peers. At this level, we make two contributions. The first contribution is a methodology that a chip processor designer can use to avoid the HIMCYFIO pitfall:

1. exposing publicly the design concepts and their trade-offs (part I),
2. recognizing and choosing audiences, and using their current expectations as a starting point to define a context (chapter 5);
3. explaining the innovation to the chosen audience(s) in the way that the audience was previously used to (chapters 6 and 7);
4. exposing and detailing how the innovation differs from past experience of the audience, within the context determined in step 2 (chapters 8 to 11);
5. providing a test and experimentation environment to third parties, and letting the chosen audience observe that third parties can use the innovation on their own (part III).

This methodology is presented throughout the dissertation, and applied to the topic of hardware microthreading as an example.

The second contribution is a discussion, spread over the entire dissertation, about what "generality" means and what features are required in computing components so that they can be advertised as general. As we argued previously in section 1.3, the future of automated computing will require humans to specialize "stem cell" computers towards specific applications. Within the context of computer architecture, the properties required from these "stem cells" have been outlined abstractly in section 1.2.1; the discussion in part II shows how to find or construct these properties in concrete implementations.



(a) Innovation process perspective (b) System stack perspective

Figure 1.3: Scope of the answers to the outer question.

The dark gray area highlights the scope of [Lan07, Lan1x].
The light gray area highlights the scope of this dissertation.

Chapter	Contributions
2. Trade-offs in microprocessor design	□□□□□
3. Architecture overview	○○○○○○○○○○○○○
4. Machine model & hardware interface	○○○○○○○○○○○○□○○○○○○○○○○○○○○○□○
5. System perspective	□□
6. Programming environment	○□□□□ ▷ ▷□□□ ▷□ ▷ ▷
7. Disentangling memory and synchronization	□□□□□ ▷ ▷
8. Visible synchronizer windows	□□□□□□□ ▷
9. Thread-local storage	□□ ▶□□ ▷
10. Concurrency virtualization	□■□ ▷ ▷ ▷ ▷
11. Placement and platform partitioning	○■□□□ ▷
12. Issues of generality	□○□□□□□ ▷
13. Core evaluation	□ ▷■ ▷ ▷ ▷□□ ▶
14. System-level issues	▶□ ▶ ▷□ ▷ ▷ ▶ ▷

Table 1.2: Overview of technical contributions per chapter.

Part I

Characterization of hardware microthreading

—Exploring the designer's mind

Chapter 2

Trade-offs in microprocessor design

Abstract

Innovation by hardware architects takes place in a cultural context, the *zeitgeist* of the current technical age. To help audiences recognize the innovation as such, this background knowledge must be identified and communicated explicitly in a way that highlights current shortcomings. In this chapter, we propose such a perspective to motivate the innovation described in chapter 3 onward. By analyzing current trends and general trade-offs in CMP design, we identify the innovation space for microprocessors with more numerous, smaller general-purpose cores featuring HMT and hardware-supported concurrency management.

Contents

2.1 Introduction

A condensed summary of the pre-2000 history of general-purpose microprocessor design is best quoted from [RML+01]:

> *In the past several decades, the world of computers and especially that of microprocessors has witnessed phenomenal advances. Computers have exhibited ever-increasing performance and decreasing costs, making them more affordable and, in turn, accelerating additional software and hardware development that fueled this process even more. The technology that enabled this exponential growth is a combination of advancements in process technology, micro-architecture, architecture, and design and development tools. While the pace of this progress has been quite impressive over the last two decades, it has become harder and harder to keep up this pace. New process technology requires more expensive megafabs and new performance levels require larger die, higher power consumption, and enormous design and validation effort. Furthermore, as CMOS technology continues to advance, microprocessor design is exposed to a new set of challenges. In the near future, micro-architecture has to consider and explicitly manage the limits of semiconductor technology, such as wire delays, power dissipation, and soft errors.*

The authors of this paper detail the obstacles faced by architects on the way to faster and more efficient processors. A summary can be found in [BHJ06a]: fundamental energy and logic costs hinder further performance improvements for *single instruction streams*. To "cut the Gordian knot," in the words of [RML+01], the industry has since (post-2000) shifted towards multiplying the number of processors on chip, creating increasingly larger CMPs by processor counts, now called *cores*. The underlying motivation is to *exploit higher-level parallelism* in applications and distribute workloads across multiple processors to increase the *overall throughput* of computations[1].

This shift to multi-core chips has caused a commotion in those software communities that had gotten used to transparent frequency increases and implicit Instruction-Level Parallelism (ILP) without ever questioning the basic machine model targeted by programming languages and complexity theory. "The free lunch is over" [Sut05], and software ecosystems now have to acknowledge and understand explicit on-chip parallelism and energy constraints to fully utilize current and future hardware.

This may seem disruptive when most textbooks still describe computers as a machine where the processor fetches instructions one after another following the control flow of *one program*. Yet this commotion is essentially specific to those traditional audiences of general-purpose processors. In most application niches, application-specific knowledge about available parallelism has long mandated dedicated support from the hardware and software towards increased performance: scientific and high-performance computing have long exploited dedicated Single Instruction, Multiple Data (SIMD), Multiple Instruction, Multiple Data (MIMD) and Single Program, Multiple Data (SPMD) units, embedded applications routinely specialize components to program features to reduce logic feature size and power requirements, and server applications in datacenters have been optimized towards servicing

[1] While parallelism can also be used to reduce the latency of individual programs, Amdahl's law gets in the way and Gustafson's law [Gus88] suggests that the problem sizes of the parallel sections instead expand to the parallelism available at constant latency.

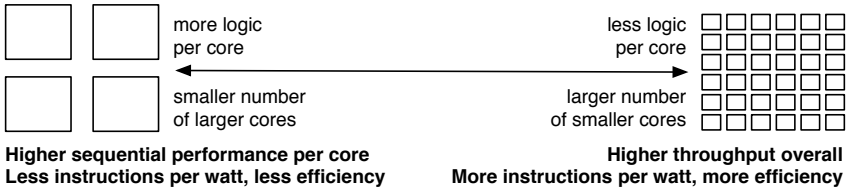


Figure 2.1: Choice between smaller or larger cores at equal logic and energy budget.

independent network streams, exploiting dedicated I/O channels and HMT for throughput scalability.

Moreover, we would like to propose that while general-purpose programmers have been struggling to identify, extract and/or expose concurrency in programs during the last decade, a large amount of untapped higher-level parallelism has appeared in applications, ready to be exploited. This is a consequence of the increasing number of features, or *services* integrated into user-facing applications in the age of the Internet and ever-increasing support of computers for human activities. For example, while a user's focus may be geared towards the decoding of a film, another activity in the system may be dedicated to downloading the next stream, while yet another may be monitoring the user's blood nutrient levels to predict when to order food online, while yet another may be responsible for backing up the day's collection of photographs on an online social platform, etc.

Even programs that are fundamentally sequential are now used in applications with high-level parallelism at scales that were unexpected. For example, the compilation of program source code to machine code is inherently mostly sequential as each pass is dependent on the previous pass' output. However, meanwhile entire applications have become increasingly large in terms of their number of program source files, so even though one individual compilation cannot be accelerated via parallelism it becomes possible to massively parallelize an entire application build.

In other words, while Amdahl's law[2] stays valid for individual programs, we should recognize that Amdahl did not predict that *single* users would nowadays be routinely running so *many* loosely coupled programs simultaneously. Hence the necessary question: *assuming that multi-scale concurrency in software has become the norm*, what properties should we expect to find in general-purpose processor chips?

We explore this question in the rest of this chapter.

2.2 Size matters

Once the architect considers a CMP design, an opportunity exists to choose how much logic to invest *per core* vs. how much logic to invest towards a *larger number of cores*.

Once concurrency is available in software, it becomes advantageous to scale back the number of transistors per core and increase the number of cores. There are two reasons for this. The first reason is that the logic and energy costs invested in individual cores towards improving sequential performance, namely larger branch predictors, larger issue

[2]Amdahl explained in [Amd67] that that the performance of one program will stay fundamentally limited by its longest chain of dependent computations, i.e. its *critical path*, regardless of how much platform parallelism is available.

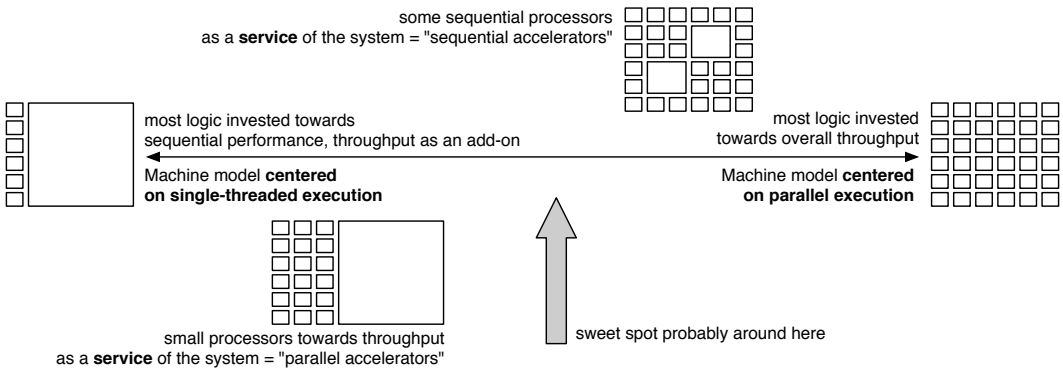


Figure 2.2: Room for sequential performance at equal area and energy budget.

width, duplicated functional units, deeper pipelines, larger reorder logic, etc., scale disfavorably with the Instructions Per Cycle (IPC) gains (e.g. two times more transistors does not increase IPC by two). Pollack summarized this situation in [Pol99] by stating that the performance of a sequential processor improves with the square root of the number of transistors. The other reason is that power consumption grows as a super-quadratic function of frequency [RML+01, SA05]: if a design enables throughput scalable with the number of cores, energy efficiency gains can be expected by running more cores at a reduced frequency (e.g. 10 cores at 200MHz each instead of 1 core running at 2GHz will consume less power, for the same maximum IPC). We illustrate this design spectrum in fig. 2.1.

2.3 Sequential performance and the cost of asymmetry

We suggest above that "smaller is better" and that many small cores will fare better overall than few larger cores in our emerging era of ubiquitous concurrency in software. Yet we should acknowledge that some inherently sequential workloads will still matter in the foreseeable future, both from legacy software and those applications where no parallel or distributed algorithms are yet known. To support these while still taking advantage of the available software concurrency, the processor architect has two recourses. The conservative approach is to favor homogeneity and slide the design cursor more to the left in fig. 2.1. This is the approach taken e.g. with the Niagara T4 [SGJ+12]. This simplifies the machine model exposed to programmers, but comes at the cost of less efficiency for more concurrent workloads.

The other approach is to introduce *static heterogeneity* and allocate some areas of the chip towards throughput and others towards sequential IPC. This is the approach taken e.g. with the AMD Fusion architecture [Adv], where "accelerator" cores are placed next to general-purpose cores on the same die. Our illustration of the corresponding spectrum of possible heterogeneous designs in fig. 2.2 reveals a possible pitfall: the appearance of *model asymmetry* as an historical artifact.

Indeed, the shift towards more on-chip parallelism has emerged from a background culture where a chip was a single processor. The availability of on-chip parallelism may thus appear as an *extension* of the well-known single processor. However, if a CMP is considered as a mostly-sequential processor with optional "parallel accelerators," this will encourage

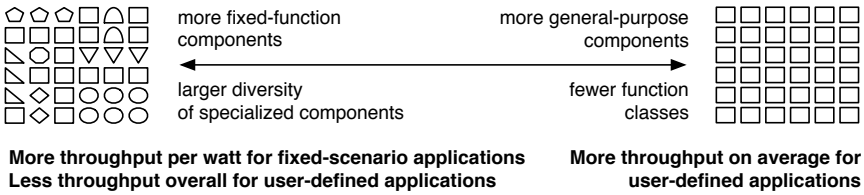


Figure 2.3: Choice between specialized functions or general-purpose cores at equal logic and energy budget.

software ecosystems to keep their focus on the overall sequential scheduling of workloads. An *opportunity loss* ensues: the design of truly *distributed applications on chip*, which would consider both throughput-oriented and sequential IPC-oriented cores as various *services* in the system that must be shared between applications, is thereby discouraged.

□ A road to avoid this pitfall is perhaps to place the focus on the *protocols* that distribute and coordinate workloads between cores on chip, and research how to build protocols that erase the performance asymmetry from the conceptual models offered through programming interfaces.

2.4 Specialization is an orthogonal, limited process

Another form of static heterogeneity already in use is one of *function*: when specific computations are provided by dedicated hardware logic. For example, a hashing unit to support cryptography and a Floating-Point Unit (FPU) for arithmetic are fixed-function components. The motivation for such specialization is the increased throughput per unit of area and per watt *for the functions specialized* and the algorithms that use them.

This opens a spectrum of designs which we illustrate in fig. 2.3. When a chip is dedicated to a given *application scenario*[3], and the scenario has been profiled, it becomes meaningful to provision the main computational components of the scenario using specialized hardware. Conversely, a given selection of specialized features is *specific* to the scenario considered, and may not match another scenario which uses different computational components.

In general, *feature specialization* in a hardware design is an orthogonal way to increase efficiency, i.e. increase overall application throughput at constant cost or reduce cost at constant throughput, *for given application scenarios*.

In contrast, a general-purpose chip is by definition not specialized towards specific scenarios. While one may be tempted to suggest mandatory specialized circuits for those few features that have become prevalent across most fields of computing, for example floating-point arithmetic, cryptography kernels and signal processing SIMD units for audio/video stream conversion, the question would still exist of *how much* logic to invest into these specialized units as opposed to e.g. more general-purpose cores, larger on-chip memories or a faster interconnect. Again, the proper methodology to strike the right balance is to consider the profile of contemporary applications at the time the overall chip design is decided and specialize for that.

Here we are able to recognize a limit on specialization.

[3]application scenario: a set of applications used in a specific pattern, for example a multimedia decoder for a television.

In general-purpose applications, workloads enter and leave the system at unpredictable times. The sharing of hardware components between workloads thus requires *on-line, dynamic chip resource management*. To satisfy the need for allocation times within the scale of operation latencies, resource management must be supported in hardware [KJS+02]. Since the amount of state that can be maintained locally on chip is limited, the component models used by on-line resource managers must be kept simple, which in turn implies that the *diversity of component properties* is kept low. An example of this can be found in the the replacement of multiple bus hierarchies in larger CMPs and Systems-on-Chip (SoCs) by standardized low-latency protocols on a common packet-switched NoC [HWC04]. Moreover, once application requirements increase in complexity, for example when starting to account for time/energy budget allowances [MMB07], the pressure to reduce component diversity to keep on-line resource managers fast increases further.

□ These observations push chip designers in two directions. One calls for the full integration of *re-configurable logic*, e.g. FPGAs, in general-purpose chips, so that functions can be specialized *on demand*. Unfortunately, practitioners have not yet come up with *update protocols* that can perform reconfiguration *at a fast rate*. The other direction pushes for *adaptive general-purpose cores* connected by a general-purpose NoC, which simplifies on-line resource management by making pools of resources *fungible*[4]. For example, a group of many in-order RISC cores on a mesh interconnect with configurable frequency and voltage may be an advantageous replacement for a fixed-frequency specialized SIMD unit with dedicated data paths, because it is reusable for other purposes without the overhead of hardware reconfiguration.

In short, while the throughput/cost ratio may be higher for isolated workloads running on specialized, monolithic units, fungibility of function is more desirable for dynamic, distributed and unpredictable workloads: it increases predictability of cost and responsiveness by reducing contention, either on the specialized components themselves or on resource managers. A recent acknowledgement of this view can be found in [SCS+08].

Note that favoring general-purpose units does not imply mandating homogeneity in chip designs: each unit can be configurable dynamically based on load and application requirements, at a fine-grain, with different throughput and cost parameters; frequency and voltage are examples. Besides constraints on on-chip resource management, outlined above, there are other simpler reasons to invest logic towards larger numbers of general-purpose cores instead of specialized functions. For one, it simplifies code generators by reducing the diversity of operation interfaces in the Instruction Set Architectures (ISAs). It also increases opportunities for fault tolerance by allowing more replacement candidates for faulty units. Finally, it reduces hardware design costs by allowing tiling.

2.5 Dynamic heterogeneity and unpredictable latencies

Another trend that supports re-configurable or fungible computing resources on chip is the increasing number of faults as the density of circuits and the number of transistors increases.

Both transient and permanent faults can be considered. Transient faults are caused mostly by unexpected charged particles traversing the silicon fabric, either emitted by atomic decay in the fabric itself or its surrounding packages, or by cosmic rays, or by impact from atmospheric neutrons; as the density of circuits increases, a single charged particle will impact

[4] *Fungibility* is the property of a good or a commodity whose individual units are capable of mutual substitution. Examples of highly fungible commodities are crude oil, wheat, precious metals, and currencies.

more circuits. Permanent faults are caused by physical damage to the fabric, for example via heat-induced stress on the metal interconnect or atomic migration. While further research on energy efficiency will limit heat-induced stress, atomic migration unavoidably entails loss of function of some components over time. This effect increases as the technology density increases because the feature size, i.e. the number of atoms per transistor/gate, decreases.

To mitigate the impact of faults, various mechanisms have been used to hide faults from software: redundancy, error correction, etc. However, a fundamental consequence of faults remains: as fault tolerance kicks in, either the *latency changes* (e.g. longer path through the duplicated circuit or error correction logic) or the *throughput changes* (e.g. one circuit used instead of two).

To summarize, the increasing number of faults is a source of unavoidable *dynamic heterogeneity* in larger chips. Either components will appear to software to enter or leave the system dynamically, for example when a core must stop due to temporary heat excess, or their characteristics will appear to evolve over time *beyond the control* of applications.

□ This in turn suggests a distributed model of the chip (independently of the suggestion from section 2.2) which can account for the transient unavailability or dynamic evolution of parts of the chip's structure.

2.6 Fine-grained multi-threading to tolerate on-chip latencies

The increasing disparity between the chip size and the gate size causes the latency between on-chip components (cores, caches and scratchpads) to increase relative to the pipeline cycle time[5]. This divergence is the on-chip equivalent of the "memory wall" [WM95]. This causes increasing *mandatory waiting times* in individual threads. Moreover, these latencies will be *unpredictable*, due to overall usage unpredictability in general-purpose workloads, due to faults as explained above in section 2.5, and due to feature size variability as the gate density increases [BFG+06]. These latencies must be tolerated by overlapping computations and communications *within* cores before the expected throughput scalability of multiple cores can be achieved.

These latencies cannot be easily tolerated using superscalar issue or VLIW, for the reasons outlined in section 2.2 and [BHJ06a]. To summarize, increasing the amount of concurrency in single-threaded cores via superscalar execution mandates non-scalable complexity in coordination structures like register files. In turn, VLIW is sensitive to mispredicted branches and variable memory latencies, and thus requires energy-inefficient speculation to maximize throughput.

An alternative to *exploit mandatory waiting times* of individual threads is Hardware Multi-Threading (HMT), i.e. the interleaving of fine-grained threads in the cores' pipelines via a hardware scheduler. Note that hardware multithreading does not increase the sequential performance of individual threads; instead, it makes a system generally more efficient (less wasted time/energy), and *increases overall throughputs* when algorithms can successfully fill waiting times of some threads by useful work from other threads without increasing the length of the overall critical path.

The general principle of interleaving multiple threads through a single processor with smaller time slices than the communication latency they should tolerate is not new [Bem57,

[5] At constant wire length, the sections above suggest more smaller cores at lower frequency, which would imply less communication delay relative to the cycle time. However, the wire delay increases relative to the transit time across a gate; the latter in turn constrains frequency and puts a lower bound on pipeline cycle time.

Sal65, RT74]. However, as the ratio between average on-chip latencies between components and pipeline cycle time grows, all inter-component events become candidate for latency tolerance, including memory operations, floating-point operations and inter-thread synchronization events. In this setting, the latency overhead of software-directed preemptive interleaving of threads over a single physical execution context, i.e. the cost of saving and restoring PC and per-thread registers, could be too large compared to the latencies to be tolerated; therefore, hardware-directed scheduling over multiple physical thread contexts must be used if all latencies must be tolerated.

There have been previous successful approaches to implement HMT in general-purpose designs. On *barrel* processors (e.g. on the CDC 6600 [Tho65] where it was first implemented, and later with Denelcor's HEP [Smi81]), the hardware scheduler assigns equal time slots to all threads. The MTA [BCS+98, SCB+98] follows up on the barrel processor concept and makes scheduling dynamic: only instructions from threads ready to execute enter the pipeline. On Simultaneous Multi-Threading (SMT) superscalar processors [TEL95, MBH+02], unused issue slots are filled by instructions from a secondary Program Counter (PC).

There are two issues with previous approaches to HMT however. One is throughput flexibility: the overall throughput should be divided more or less equally between active threads. This is an issue with pure barrel processors: with N contexts, the performance of each threads is $1/N$ even if there are less than N threads active. This has been addressed in the MTA's dynamic schedule queue, and is naturally not an issue in processors primarily designed for sequential performance. The other is robustness to branches: with long pipelines, the cost of mispredictions (or branches taken if there is no prediction) is high, as the pipeline must be flushed. This is an issue with the 21-stage MTA pipeline and most contemporary SMT superscalar designs.

□ These observations suggest that the benefits of HMT are anticiped to be apparent with shorter pipelines and dynamic schedule queues.

2.7 Pressure for hardware-assisted concurrency management

Assuming CMPs with an increasing number of cores and per-core HMT, *space scheduling* must be implemented to spread concurrent software workloads to the chip's parallel execution resources. Space scheduling can be done either in software or in hardware.

With a software scheduler, each hardware thread is controlled by a program that assigns tasks using state taken from main memory. Specifically, each hardware thread is controlled by an instance of a software scheduler in an operating system, which is notified, via interrupt signalling and memory-based communication, upon task creation and begins execution until a termination or suspension event. This can occur even when thread interleaving is performed at a fine grain in hardware. It is also relevant even when there is no need for time sharing of multiple tasks onto a single hardware thread, for example when the number of tasks is smaller or equal to the number of hardware threads, or when cooperative scheduling is sufficient. However, the choice of a software scheduler assumes that the workload per task is always sufficient to compensate the non-local latencies incurred by memory accesses to task state during schedule decision making and task assignment.

This assumption traditionally holds for coarse-grained concurrency, for example external I/O. It can also hold for regular, wide-breadth concurrency patterns extracted from sequential tight loops, via blocking aggregation (e.g. OpenMP). However the situation is

not so clear with fine-grain heterogeneous task concurrency. For example, graph transformation and dataflow algorithms typically expose a large amount of irregularly structured, fine-grained concurrency. In these cases, a strain is put on compilers and run-time systems: they must determine the suitable aggregate units of concurrency from programs that both optimize load balancing and compensate concurrency management costs.

□ This motivates the acceleration of space scheduling, considered as a *system function*, using dedicated hardware logic. This idea to introduce *hardware support for concurrency management* is not new; it was pushed by researchers until twenty years ago [Smi81, HF88, NA89, MS90, CSS+91]. Back then, it met with resistance against the introduction of explicit concurrency in applications. Now that on-chip software concurrency is the norm, hardware support deserves renewed attention for two reason.

One is the potential gain in resource fungibility obtained by the replacement of specialized SPMD/SIMD units by multiple general-purpose cores (cf. section 2.4). For this to be tractable, the overhead to dispatch an SPMD task over all participating pipelines must be comparable to or smaller than the latency of the operation, e.g. a couple dozen pipeline cycles for most SPMD workloads. To make general-purpose cores an attractive substitute to specialized SPMD/SIMD units, extra hardware support must exist with low-latency bulk work distribution and synchronization.

The second argument is cost predictability: when a software scheduler is involved, it competes with algorithm code for access to the memory components. The overhead of communication and synchronization between software schedulers for task management increases with the number of hardware threads and interferes with communication for computations, introducing jitter [ALL89]. This can be avoided by a dedicated task control network separate from the memory network.

We should acknowledge here that hardware task and thread management somewhat reduces flexibility in the definition of a task from the perspective of software. As noted in [LH94], "thread definitions vary according to language characteristics and context-switching criteria." We should not consider this argument to be a ban on hardware support for concurrency management, however. In fact, it seems to us desirable to provide some form of *general* hardware/software interface for the definition of concurrent workloads, so that hardware components with different designs can implement them in a heterogeneous CMP without creating asymmetry in the machine model exposed to software.

Summary

□ In this chapter, we have identified the design trade-offs available to a CMP designer in the age of ubiquitous software concurrency. Our analysis suggests chip designs towards smaller, simpler general-purpose cores where larger cores optimized towards sequential performance serve as a *service* to the rest of the system. We have also identified the applicability and limitations of hardware specialization of specific application features. We recognized Hardware Multi-Threading (HMT) as a means to tolerate unpredictable on-chip latencies and identified the renewed relevance of hardware-supported concurrency management for on-chip parallelism. This argument updates and extends previously published justifications [BJM96, BHJ06a] for architecture research towards hardware microthreading.

Chapter 3

Architecture overview

Abstract

Here we provide an overview of the architectural concepts that characterize hardware microthreading, and outline the architecture design principles which guide its implementation in hardware. We also present the implementation choices that have been made prior to our work.

Contents

3.1 Introduction

In the previous chapter we have outlined an innovation space for CMPs with smaller, more numerous general-purpose cores, a distributed structure, focus on inter-core protocols, and HMT to tolerate diverse on-chip latencies. The work on hardware microthreading at the University of Amsterdam, performed mostly prior and outside of the scope of our own work, can be describe as a design activity towards new CMP structures in this innovation space.

The outcome of these activities is an overall CMP architecture model with custom on-chip components. Yet previous reports on this activity (in [BHJ06a, Has06, BHJ06b, Lan07, BGJL08, JLZ09a, JLZ09b, Lan1x] and other academic publications from the same authors) have focused mainly on general motivations and the eventual applicability of the proposed architecture, without providing comprehensive descriptions of the architecture itself in a way amenable to study by outsiders to the field, in particular software communities. The reason for this is that the academic communities around computer architecture are traditionally competing via performance results, leaving little publication space for describing components and design trade-offs.

To increase the visibility of the proposed innovation to external audiences, and provide a technical context to the remainder of our dissertation, the present chapter describes the general principles of hardware microthreading and its possible implementations. Together with chapter 4 and the accompanying Appendices A to E, this *contemplative description* constitutes our first contribution. This chapter is intended to complement [Lan07, Lan1x], which detail the design of the hardware implementation which we used as our main reference platform.

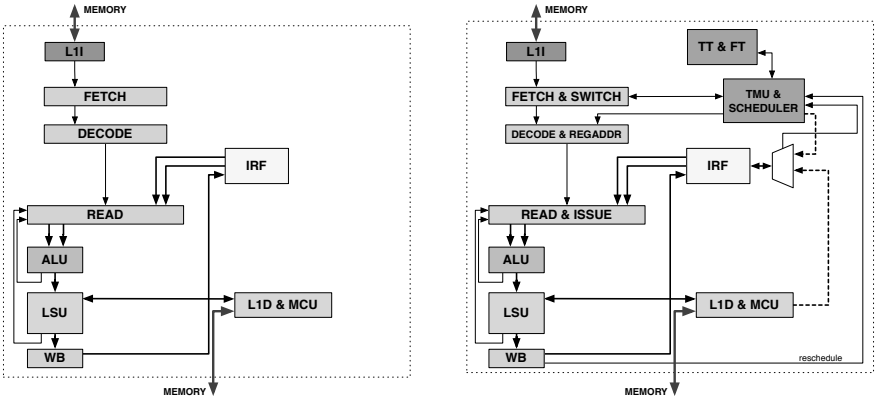
3.2 Core micro-architecture

The micro-architecture of individual cores favors simplicity, i.e. fewer gates per core to increase the core count on chip and the performance per watt, over sequential performance per core as suggested in section 2.2. It also combines dynamically scheduled HMT for latency tolerance with a short pipeline to tolerate branches, as suggested in section 2.6.

3.2.1 Multithreading and dataflow scheduling

○ The design proposes to extend an in-order, single-issue RISC pipeline as depicted in fig. 3.1:

- the *fetch stage* is extended to use PCs from a *thread active queue* (FIFO), provided by a *thread scheduler* in a Thread Management Unit (TMU). Control bits in the instruction stream read from the L1 I-Cache ("L1I" in the diagram) indicate when to switch to the next PC in the queue. A switch occurs also whenever the fetch unit starts to read from a different line in the I-Cache in order to ensure that no executing PC misses. If no PC is available, the fetch stage becomes idle;
- the *decode stage* is extended to translate the register names listed in the instruction codes by a *register window offset* provided by the TMU for the current thread. This provides dynamic addressing in the Integer Register File (IRF) for different threads, as well as dynamic sizing of the register window in each thread;
- the *Register File (RF)* is extended with *dataflow state bits* on each register, which indicate whether a register is full (has a value), empty (no value) or waiting/suspended (a value is expected, a thread is waiting on a value). Bypasses from the Arithmetic



(a) Original RISC pipeline.

(b) Pipeline equipped with a Thread Management Unit (TMU) and dataflow scheduling.

Figure 3.1: Microthreaded extensions on a typical in-order, single-issue 6-stage RISC pipeline.

Logic Unit (ALU) and Load-Store Unit (LSU) back to the read stage are also extended to carry the dataflow state of updated registers;

- the *read stage* is extended to check the dataflow state bits, and suspend the thread when a register operand is not full;
- the *Memory Control Unit (MCU)*, actually part of the LSU but separated in the diagram for clarity, is extended to set the target register to the waiting state upon L1 read misses, but *without stalling the pipeline nor suspending the thread* as the next instructions may be independent[1]; meanwhile, load completions are written asynchronously to the register file and wake up threads, as discussed below and in section 3.2.2;
- the *writeback stage* is extended to also propagate control signals from the thread management instructions (discussed further in chapter 4) to the TMU, and reschedule the PC of the current thread on the active queue if it was descheduled at the fetch stage but has not suspended in the pipeline;
- the newly introduced *Thread Management Unit* maintains state information for threads (PC, register window offsets, etc.) in dedicated structures on the core.

The notion of "dataflow scheduling" comes from the fact that threads are suspended upon reading from empty operands, and resume execution only once the operand becomes available. This asynchrony is implemented by storing the head and tail of a list of suspended threads in the input register when it is not full. When an asynchronous operation completes (e.g. a memory load), the register is updated by the corresponding unit and simultaneously the list of suspended thread(s) is appended to the active queue. Each thread context is described by an entry in a dedicated memory, called the *thread table*, which contains its PC and *next* field for schedule and suspend queues. An example is given in fig. 3.2, which illustrates the thread table after threads 5, 2, 3, 1 have tried to read from a waiting synchronizer.

[1]That is, the thread does not suspend on the load itself and independent subsequent instructions are allowed to execute. If some subsequent instruction is data dependent on the missed load, that instruction will suspend the thread if its operands are not ready yet.

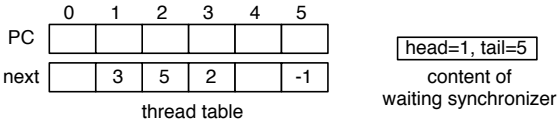
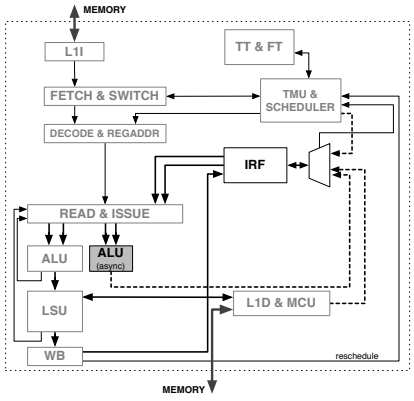


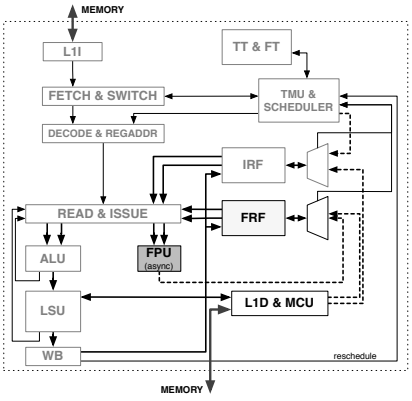
Figure 3.2: Example suspended thread list.

Side note 3.1: Active, ready and waiting queues.

For the purpose of a high-level overview it is sufficient to consider a single queue for schedulable threads. However a naive implementation based on a single queue would be ill-equipped to deal with I-cache misses. Instead, the detailed design proposes to place schedulable threads on a *ready queue*, and only move threads from the ready queue to the active queue (connected to the fetch unit) once their instruction data is in the I-cache. Upon I-cache read misses, ready threads are placed in a *waiting queue* instead, and moved to the active queue upon completion of the I-cache read. See also side note 3.3, [Lan07, Sect. 4.3] and [Lan1x, Sect. Thread Scheduler].



(a) Microthreaded pipeline with an asynchronous ALU.



(b) Microthreaded pipeline with an asynchronous FPU.

Figure 3.3: Using asynchronous FUs for latency tolerance.

The scheduling strategy is not purely dataflow oriented however. To avoid pipeline bubbles, the code is statically instrumented by control bits, visible at the fetch stage, which indicate whether an incoming instruction *may* suspend. If so, and there are more threads in the active queue, the current thread is popped from the active queue (descheduled) after the current instruction is fetched, and the fetch unit switches preemptively to the next thread's PC for the next cycle. This effectively achieves *zero-cycle thread switching*.

This overall core structure requires only two read ports to the IRF for the read stage, one read/write port for the writeback stage and one arbitrated read/write port for asynchronous completions.

3.2.2 Long latency operations and completions

○ As mentioned previously, upon L1 read misses the MCU writes back memory load completions asynchronously to the register file via a dedicated port. To determine which registers

to update upon completion, the LSU stores the address of the register to update in the L1 cache line[2].

Dataflow scheduling can also be used for multi-cycle local operations. For example, integer multiply and divide can be implemented as asynchronous integer Functional Units (FUs) which write back their result asynchronously to the register file (fig. 3.3a), waking up any suspended thread(s) during the write if any. The same principle can be used again for an FPU, as depicted in fig. 3.3b. The Floating-point Register File (FRF) is equipped with dataflow state bits as well, and Floating Point (FP) registers are used in the same way as integer registers for the purpose of scheduling. Since the FUs are thereby fully asynchronous, it becomes possible to share an FU between multiple cores; this opportunity is exploited in the configuration studied in chapter 13. Note that despite the scalar structure, there is little contention on the asynchronous R/W port of the register file because at most one instruction is issued per cycle, and thus there is at most one completion per cycle on average [HBJ07].

In all these scenarios, the *continuation* of the long-latency operation, represented by a list of threads waiting on the value, is stored in the target register until the operation completes. To avoid losing this continuation and causing deadlock, any further instruction with the same register as its output operand will suspend as well or stall the pipeline until the first long latency operation completes[3].

3.2.3 Threading to handle pipeline hazards

○ A perhaps striking feature of the proposed design is the absence of several components found in other processors to handle pipeline hazards.

As seen above, the design self-synchronizes true dependencies. Out-of-order completions from asynchronous FUs do not need reorder buffers, because further dependent instructions (both true dependent and output dependent) will self-synchronize in instruction stream order. A branch predictor becomes unnecessary once tight loops are replaced by dependent threads, one per loop iteration, interleaved in the pipeline. Branches are also marked to cause a switch upon fetch, so that instructions from other threads can fill the pipeline slots immediately following a branch while it is resolved.

There are possible structural hazards, each with candidate solutions. A store to an L1 line waiting on a load may stall to preserve the ordering of memory updates. This can be solved by storing the continuation of the load in a separate structure than the line itself and merging stores with the incoming line upon completion. An instruction issued to a busy non-pipelined long-latency FU, such as FP divide, may stall if there is no space left in its input buffer(s). The alternative is to increase the number of FUs. A memory operation reaching the MCU from the pipeline at the same time as a memory completion may stall until the completion is effected. The solution is to dual port the cache and arbitrate updates at the granularity of single lines. In any case, solutions exist to avoid wasting the pipeline slot and rescheduling the thread. While these design choices fall outside the scope of our work, we mention them here for clarity and completeness.

[2]Specifically, the L1 line contains the address of the head of a linked list of registers to update, since multiple loads can be waiting on the same line.

[3]Whether this situation causes a stall or a suspension is still a topic for research, but is outside the scope of our work. In either case the architecture should ensure that continuations are not lost. In the early implementations, this guarantee was not provided, but this was later fixed, cf. section 4.6.1.3.

3.2.4 Critical latencies, IPC and throughput

○ The proposed design issues at most one instruction per cycle; the throughput is determined by frequency, pipeline utilization and the number of dataflow misses, i.e. reads from empty operands.

The two structures with non-trivial access latencies in the critical path of the pipeline cycle time are the register file and the L1 D-cache. While the number of ports on the register file is low, the number of registers increases with the desired number of simultaneously allocated threads. The size of the L1 D-cache, and thus its access latency, increases as the desired heterogeneity of workloads increases.

To quantify the impact of these structures and predict the maximum allowable core frequency, CACTI [WJ96] estimates using conservative parameters have been computed. For example, given a 64-bit core implementation configured with 1024 registers per RF, a 4-way associative 4KB L1 D-cache, and support for up to 256 threads, the maximum allowable pipeline frequency at 65nm CMOS lies in the range 800MHz to 1.5GHz; we used these parameters during our evaluation activities, with a 1GHz pipeline frequency. At 45nm CMOS, the pipeline frequency can be increased to 2GHz.

3.3 Concurrency management within cores

The design provides dedicated hardware circuits to accelerate the organization of software concurrency during execution, as suggested in section 2.7. We explain this below.

3.3.1 Thread management

○ In most processors, the program counter is initialized at hardware start-up to a predefined value, and from then on updated either by branches, traps or interrupts. The traditional vision is characterized by the assumption that the processor is dedicated to an instruction stream as long as it is powered. The logical activation and de-activation are indistinguishable from the physical events "start of world" and "end of world," which cannot be controlled in software. This abstraction has been carried over to recent multithreaded architectures, for example Niagara and HyperThreaded processors, where each thread of execution is activated upon core initialization and subject to traps and interrupts in the same fashion as a sequential processor.

In contrast to this, microthreading reifies "thread creation" (processor start up) and "thread termination" (processor shut down) as hardware events that can be programmed. The initial configuration of the PC, which in a traditional architecture is fixed statically at design, is also configurable dynamically during creation. However, the choice of which hardware resources (registers, PC slot) to use is not directly programmable; they are allocated dynamically by the TMU upon receiving a thread creation event. As we describe below, the arguments of the event determine the initial PC, as well as how many registers to allocate.

This allows us to compare a conventional approach to hardware multithreading (fig. 3.4) with the microthreaded approach (fig. 3.5). In the conventional case, the thread is ready to execute instructions from a pre-configured PC as soon as the chip is powered. The thread may suspend upon blocking operations, or upon executing the HALT instruction which stops processing entirely. When halted, only an interrupt can wake the thread up and make it usable again. With microthreading, thread contexts are initially "empty," meaning unused and not mapped to any workload. Upon receiving a *thread creation event* the TMU allocates

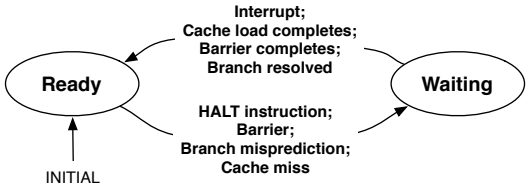


Figure 3.4: Thread states in the Niagara architecture.

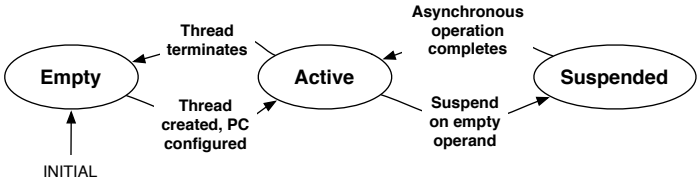


Figure 3.5: Thread states in the microthreaded architecture (simplified).

See also the detailed diagram in fig. 3.6.

Side note 3.2: New instruction vs. control bits for thread termination.

In an alternative implementation, a dedicated "terminate thread" instruction could be added to the ISA. However, since the architecture already uses out-of-band control bits for scheduling (section 4.4), these can be extended to provide control over thread termination without the need for a new instruction. This choice can also increase pipeline utilization for small threads with only a few instructions each: control bits do not require occupation of a pipeline slot, and threads can terminate and be cleaned up as a side-effect of their last useful instruction.

a thread context from a *thread table*, populates the context with a PC and makes it "active." From that point forward the thread behaves like a sequential processor. The additional extension is that program-specified, out-of-band control bits in the instruction stream can trigger the *thread termination event*, which stops processing instructions from that thread and releases the context for another thread creation (cf. side note 3.2).

This basic thread management scheme implements the dynamic creation of *logical threads* over *physical thread contexts*, analogous respectively to tasks and worker threads in software concurrency managers. The resulting support for a *dynamically variable number of hardware threads* is a distinguishing feature of the design.

3.3.2 Preemption and asynchronous events

○ Traps and interrupts have been designed in sequential processors *essentially* as a means to multiplex access to the single-threaded processor between a program and an asynchronous event handler. When the processor has as many thread contexts as there are event channels, each event handler can run in its dedicated context, and support for control flow interrupts is not required in principle.

The microthreaded core exploits this opportunity by replacing support for external interrupts with support for a large number of thread contexts. For example, the reference implementation we use in our work supports 256 thread contexts. The reception of external events can then be implemented on a control NoC either by *active messages* [vECGS92],

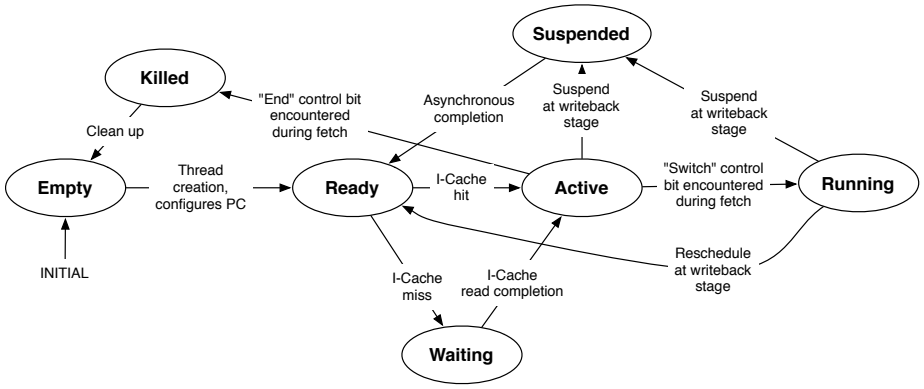
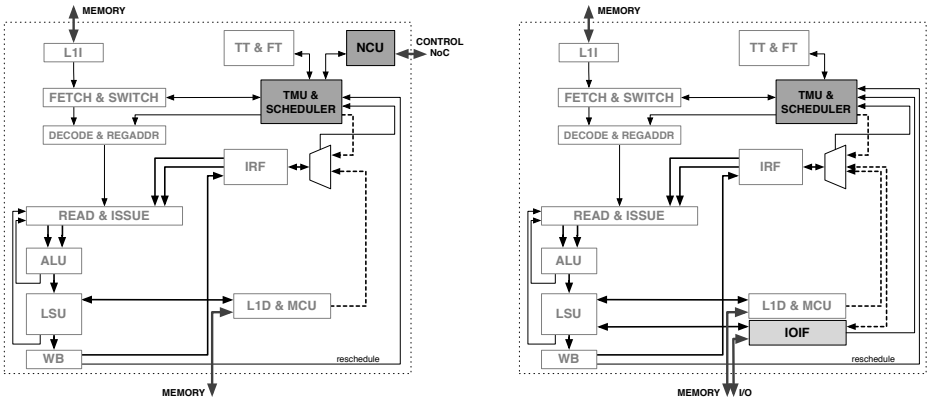


Figure 3.6: Thread states in the microthreaded architecture.



(a) Interface for active messages on a control Network-on-Chip (NoC). A Network Control Unit (NCU) disassembles incoming NoC packets to TMU control signals and *vice-versa*.

(b) I/O interface to translate generic I/O events to dataflow events and thread creation requests. The mapping between I/O channels and control signals is programmable via configuration registers.

Figure 3.7: Support for incoming external asynchronous events.

Side note 3.3: Fine-grained thread states.

The actual thread states are detailed in fig. 3.6:

- the interface to memory via an I-Cache suggests a discrimination between "ready," "active" and "waiting" states, so the latter accounts for ready threads waiting to refill the I-cache;
- the First-In, First-Out (FIFO) nature of the active queue requires the pipeline to remove active threads from the queue upon switch, and add them back at the writeback stage if they are still schedulable. This in turn requires a distinction between "active" (still in FIFO) and "running" (not in FIFO, but still in pipeline);
- to prevent a "waiting" thread from never becoming active, I-cache lines must be locked until the corresponding thread gets a chance to run some instructions. To achieve this, the following conditions are met: the I-cache is fully associative; there are more I-cache lines than pipeline stages; and each line contains a reference counter with the number of waiting/active/running threads currently using the line, so that it can be evicted only once all associated threads exit the pipeline.

In hardware, these states are not named as state bits in a memory; they can be distinguished by which hardware components currently refer to the thread entry. Performance counters can be implemented that keep track of how many threads are in each state.

which carry an entire request for thread creation (PC, optional value argument) directly to the TMU, or by an *I/O interface*, which translates interrupt-style control signals to either dataflow events in the pipeline or thread creation events in the TMU. Both schemes are illustrated in fig. 3.7. Research is ongoing to implement *scheduling priorities* by placing threads created from asynchronous events in different schedule queues.

This *preference for active messages and thread creations to signal asynchronous events over control flow preemption via interrupts* is another distinguishing feature of the design.

3.3.3 Configurable register windows

○ In a traditional RISC, register operands in machine code index the register file physically. There have been three extensions to this in previous work. In the SPARC architecture, *sliding register windows* are implemented by updating a *Current Window Pointer (CWP)* at function call and return. The CWP in turn offsets the instruction-specified register name to index a register file containing multiple 16-register blocks [GAB+88, MAG+88]. This structure was intended to optimize the overhead of function calls by avoiding spills, and in turn enables an optimization of the physical layout, due to the fact that only one window is used at a time [TJS95]. In superscalar designs, *register renaming* [Sim00] changes register names in instructions to index a larger physical register file than the logical numbering allowed by the ISA. In hardware multithreaded processors, each thread is configured to use a private *thread register window* in a separate region of the physical register file.

These advances have introduced a distinction between the *register file*, which is the physical data store, and the *logical register window*, which is the virtual set of register *names*, i.e. virtual register addresses, that can be used by machine instructions. The translation of names to a physical address in the store is performed in the issue stages of the pipeline.

The microthreaded core exploits this opportunity further and introduces two innovations: the ability to *configure the size of the logical register window* and the ability to *configure the overlap between register windows of separate threads.*

Variable window sizes are motivated by the need to increase utilization of the register file, which is typically the most expensive resource on the core. The argument is that tolerating on-chip latencies requires concurrent workloads that are short, possibly a few instructions. This in turn entails that some threads only require a few registers, i.e. potentially many

less than the maximum window size allowable by the ISA. By mapping only the part of the logical register window that is effectively used by a short-lived thread, it becomes possible to create more threads on the same number of physical registers and RF utilization is increased.

For example, the reference configuration we use provisions 1024 physical registers and 256 thread contexts. This leaves room for 33 threads with a fully mapped 31-register ISA window, or 256 threads with only 4 registers mapped in each, or heterogeneous combinations of any intermediate configuration. A dedicated hardware *Register Allocation Unit (RAU)* in the TMU performs the dynamic allocation and de-allocation of physical registers upon thread creation and termination. Upon exhaustion, context allocation is delayed until there are enough registers available to satisfy a request.

Meanwhile, overlapping register windows enable fast thread-to-thread dataflow communication and synchronization. When two instructions in separate threads target the same physical register, the dataflow scheduler (cf. section 3.2.1) ensures that the consumer instruction is scheduled only when the producer instruction completes. Moreover, if the instructions of both threads interleave in the pipeline, the bypass bus ensures that the data flows from one thread to another without any waiting time. While this concept seems challenging to exploit in a compiler or hand-crafted program, we shall explore possible uses in chapters 4, 6 and 8 and section 13.8.

This *fine-grained register window management* combined with the exploitation of *shared registers for zero-cycle dataflow synchronization* are yet two other distinguishing features.

3.3.4 Synchronization on termination

○ Synchronization on termination, also called "waiting" or "joining," is a fundamental feature of concurrency management systems.

A common approach to implement this primitive is found in Unix, described for example in [MBKQ96, Chap. 4]: when a task terminates, it is moved from the "active" state to the "zombie" state, and its parent (if any) is notified with a signal. Independently, another task can use the "wait" primitive which suspends the waiting task until the target task terminates. When the target task terminates, the "wait" primitive completes and returns the termination status of the target task to the waiting task.

The proposed architecture design implements this synchronization in hardware. A control event to the TMU, called *request to synchronize* can associate a thread context to the address of a register. When the thread terminates, the TMU writes a value into this register. Independently, another thread which has this register mapped into its logical register window can initially set the register to the "empty" state, and then start reading from it, which causes the thread to suspend. The waiting thread only resumes execution once the register is written to, i.e. when the target thread terminates.

3.3.5 Bulk creation and synchronization

○ The features described so far are sufficiently general to support nested fork-join concurrency in programs, with threads created and synchronized one at a time. Yet the proposed design makes another step towards reducing management overheads.

Considering SIMD/SPMD workloads of a few instructions over a large data set, the overhead of explicit instructions to create and synchronize every instance of the parallel workload would not provide any benefit compared to a partially unrolled sequential loop. Instead, the TMU provides a dedicated, autonomous *Thread Creation Unit (TCU)* in hardware which

is able to create and start threads asynchronously, at a rate of one every pipeline cycle or every other cycle, initialized by a single *bulk creation* event. A bulk creation event specifies a common initial PC for all created threads and a common register window configuration, including a common overlap factor between all created windows. To distinguish the created threads, the bulk creation event also specifies a configurable range of *logical thread indexes* assigned sequentially by the TCU to each created thread, and pre-populated in a private register upon thread initialization.

For synchronization, the TMU binds a single bulk synchronizer to all the threads created from a single bulk creation event. This bulk synchronizer acts as a semaphore. This modifies the synchronization on termination described in section 3.3.4: the event "request to synchronize" binds the bulk synchronizer, and not a single thread context, to a configurable register. Only when all associated threads terminate, does the bulk synchronizer cause the TMU to write a value to the target register, and let the waiting thread wake up. This allows a thread to wait on all threads in the group in a single register read. Beyond the reduction of thread management overheads for groups of related workloads, this facility also reduces fragmentation in the RF by grouping allocation and de-allocation requests in the RAU.

This particular feature which *binds a set of threads by a common bulk creation event and a common bulk synchronization structure* is another distinguishing trait of the proposed design. Groups of bulk created threads are referred to as "families" in previous publications; we discuss this term further in section 4.2.1.

3.4 Multi-core architecture

Placing multiple simple cores together is not sufficient to guarantee scalability of throughput with the number of cores. Namely, the interconnect must cater to scalable throughput (cf. section 2.2). Also, dedicated circuits must be present to provision low-latency distribution of data-parallel workloads to multiple cores, if these cores are to become an advantageous substitute to specialized SPMD/SIMD workloads (cf. section 2.4). We explain the corresponding characteristics of the proposed CMP design below.

3.4.1 Memory architecture

○ The core design is optimized for tolerating latencies using asynchronous transactions with multiple in-flight operations, and is thus intended for use with memory systems that support either split-phase transactions or request pipelining. However, the concepts behind microthreading do not mandate a specific memory system: a designer may choose to integrate microthreaded cores with memory technology implemented independently. This was illustrated on a FPGA, where the UTLEON3 [DKK+12] microthreaded cores are connected to an industry-standard memory bus.

Regardless of which memory system is used, system-level design choices will impact the machine model significantly. First, an implementation can choose between a distributed memory architecture, where each cores sees a separate address space from all other cores. Or it might choose to use a shared address space. Then it can choose between different caching protocols. Depending on these choices, a *memory model* is shaped into existence.

The proposed architecture assumes a model based on a shared address space over multiple caches and backing stores. This in turn requires to consider a spectrum of design choices related to *cache coherency*. At one end of the spectrum an implementation may guarantee that all stores by all cores are visible to all other cores in the same order, for example by

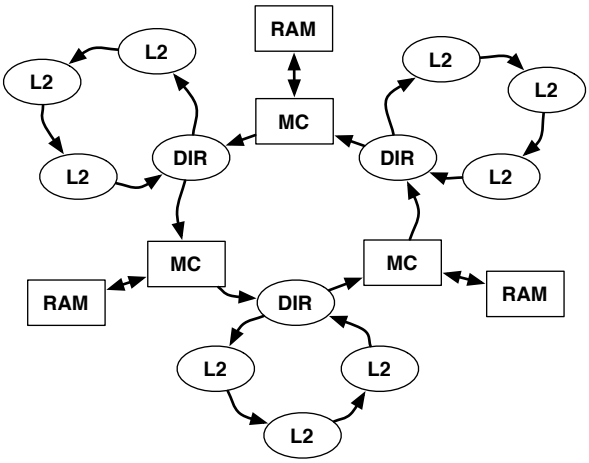


Figure 3.8: Example proposed distributed cache topology.

Multiple cores can share a single L2 cache via a local snoopy bus. "DIR" is shorthand for "cache directory"; "MC" is shorthand for "memory controller."

Side note 3.4: Using a distributed cache between microthreaded cores.

To demonstrate the usability of microthreading with larger core counts, a scalable memory system is needed. As a step in this direction, research efforts have been invested separately from our work in a dedicated synchronization-aware, distributed cache network.
The main property of the proposed distributed cache is that cache lines are automatically migrated where they are used. Concurrent loads create multiple copies of a cache line, while concurrent stores may gather a single updated copy to the location of the last store. The proposed cache architecture connects multiple cores to a shared L2 cache, and organizes L2 caches in rings. For a small-scale system (e.g. less than 64 cores) a single ring is used; for larger configurations a hierarchy of rings is used (cf. fig. 3.8). An early version of the protocol has been published [ZJ07, DZJ08] and research is ongoing on this technology concurrently to our work. It assumes memory controller(s) to external Random-Access Memory (RAM) connected to the top-level ring, and pipelined RAM interfaces, such as JEDEC's DDR[a] and Rambus' XDR[b].

[a] JEDEC Standard JESD79 and later revisions, cf. `http://www.jedec.org/`.
[b] `http://www.rambus.com/xdr`

broadcasting a lock on the cache line before each store is effected and invalidating all copies. At the other end of the spectrum, no automatic consistency is implemented in hardware, so software must control cache coherency explicitly. Our reference implementation uses a memory network of distributed caches (side note 3.4) which lies somewhat in the middle: it proposes to effect memory stores in local copies of cache data, and only propagate and merge update copies prior to bulk creation of threads or bulk synchronization on termination. With this implementation, memory can be shared consistently across bulk events, but may not be consistent between sibling threads that are not related by bulk events.

Different implementation choices thus have a dramatic impact on the memory model. In chapter 7, we revisit this topic and further explore how to make these choices independent from concurrency management issues from the software perspective.

3.4.2 Concurrency management across cores

3.4.2.1 In other multi-processor systems

Work distribution across multiple performance-oriented general purpose processors has traditionally been coarse-grained and ill-suited for fine-grained, low-latency distribution of workloads. The conventional mechanism can be described in general terms: code and data are shipped to a memory close to the core at a fixed address; this address is configured in the interrupt vector of the target processor; then an Inter-Processor Interrupt (IPI) is sent to the remote processor to trigger execution. Synchronization on termination is expensive: the thread requesting to wait from core A registers its identity in memory to a system-level scheduler in software; when a target thread on core B terminates, its local scheduler looks up any waiting thread in memory, then sends an IPI from B to A, which signals A's software scheduler; A's scheduler then looks up the origin of the IPI in memory and triggers resumption of the waiting thread.

This is the protocol used in most of the Top500[6] and Green500[7] supercomputers at the end of 2011, which are based on IBM's POWER, Intel's Xeon, AMD's Opteron and Sun's/Fujitsu's SPARC6. It is used even between multiple cores on the same chip. It is also used with most processors in embedded SoCs, including all ARM and MIPS-based cores. This protocol is due to the processors' ISA: in most ISAs only memory-related operations and interrupt instructions have the ability to communicate and synchronize with other components in the system.

This protocol is coarse-grained for three reasons. First, the overhead of a full point-to-point memory synchronization requires full cache lines to be exchanged between the initiator and the target, even if the target workload operates on independent streams of data. Then a scheduler in software on each processor must disambiguate each incoming IPI by looking up the initiator and parameters in shared structures in memory. Then the abstraction mismatch between the interrupt-based signalling mechanism and the thread-based programming model requires indirection through a software stack to effect the control events. This implies initialization and start-up overheads of hundreds if not thousands of pipeline cycles, and similar overheads again for synchronization on termination.

Incidentally, other hardware protocols for work distribution have existed and continue to be popular to this day. For example in IBM's Cell Broadband Engine, the Memory Flow Controller (MFC) inside each Synergistic Processing Element (SPE) provides a hardware request queue where the Power Processor Element (PPE), a general-purpose PowerPC core, can write new thread requests remotely. The MFC then autonomously starts execution of the thread on the Synergistic Processing Unit (SPU) [GEMN07]. Another example is NVidia's Tesla architecture. On this General-purpose GPUs (GPGPUs), the host-GPU interface streams units of work to the Texture/Processor Clusters (TPCs) via dedicated data paths in hardware. They are then decoded on the TPC and distributed to the Streaming Multiprocessors (SMs) via dedicated, autonomous SM controllers [LNOM08].

3.4.2.2 In the proposed architecture

○ Meanwhile, we have already explained in section 2.7 that further exploitation of multiple cores on chip will require lower concurrency management latencies. The proposed multi-core architecture design provides two features in this direction.

[6] `http://www.top500.org`
[7] `http://www.green500.org`

Side note 3.5: Sub-events for remote creations.

While delegation requests conceptually include all the parameters of a bulk creation, we will find useful to separate the phases in sub-events, namely *requests for allocation* of thread contexts and registers, *remote register accesses* to populate initial value in registers, *bulk configuration requests* to configure logical index ranges, and *creation requests* which trigger the actual activation of threads in the TCU.

The first feature is the ability of the TMU to *generate active messages* on the NoC, upon local reception of a *delegation request* control event from the pipeline. This complements the mechanism described in section 3.3.2 and fig. 3.7a: just as cores can receive remote thread creation events from the NoC, they can symmetrically generate those requests as well (cf. also side note 3.5). For synchronization on termination, the protocol from section 3.3.4 is also extended: a *request for remote synchronization* generated by the pipeline is forwarded by the TMU through the NCU and NoC, and is then received by the remote TMU which associates both the originating core address and register address to the target thread context. When the thread terminates, its local TMU sends a *remote register write* message through the NoC to the TMU of the processor hosting the waiting thread, which writes the value to the RF which in turn causes the thread to wake up. This feature enables fully general concurrent work creation and synchronization with an overhead of only a few cycles between adjacent cores on the NoC.

The other feature is an extension of bulk creation to support *automatic distribution of threads across cores*. In the TMU, an optional parameter can specify to restrict the number of logical thread indexes to create locally, and forward the remainder as a bulk creation request to another core. The total number of cores to use this way is also a parameter, or a feature of the NoC topology (which can be organized in static clusters). The thread creations are then effected locally by the TCUs in parallel. This way, a single bulk creation request sent to one core can trigger simultaneous thread creation on multiple cores. Similarly, a parameter can cause the TMU to duplicate automatically a register write request to all the cores participating in the bulk creation, implementing a broadcast operation. For synchronization on termination, the bulk synchronizers are *chained* from one core to another when the bulk creation request is forwarded, so that the bulk synchronization on the first core only completes when all "next" cores have synchronized locally. In short, this combination of features enables the automatic distribution of parallel workloads across cores using single requests to one core, including the parallelisation of thread creation itself. We describe cross-chip distribution further in chapter 11.

While setting up a coordinated group of threads across cores using this latter feature set may seem complicated, as we will see in chapter 4 this complexity is hidden behind simple abstractions.

3.4.3 Logical and physical networks

○ The previous sections have outlined three logical networks:

- the *memory network*, used for loads and stores issued by programs;
- the *delegation network*, used for program-generated remote concurrency management messages across arbitrary cores;
- the *distribution network*, used for TMU-controlled bulk distribution across neighboring cores.

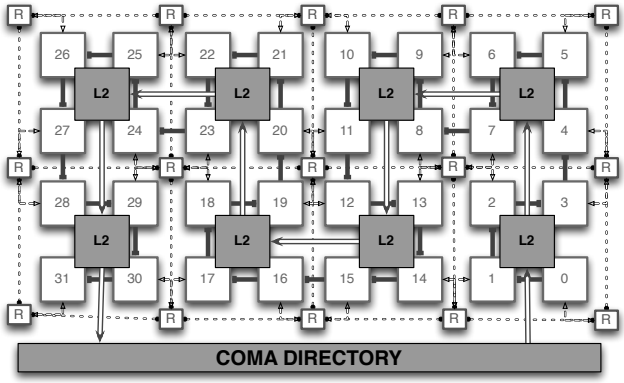


Figure 3.9: 32-core tile of microthreaded processors.

White numbered tiles represent cores. "R" tiles represent routers for the narrow delegation mesh. The thick black link between cores represents the linear bulk distribution network.

Event name	Parameters
thread creation (subsumed by bulk creation)	PC, register window layout
request for synchronization (subsumed by bulk sync.)	target thread context, address of register to write to upon termination
bulk creation	common PC, common register window layout, overlap factor, logical thread index range
request for bulk synchronization	target bulk synchronizer, address of register to write to upon termination
remote register reads & writes	target register address, value (for writes), address of register to send value to (for reads)

Table 3.1: Control events to the TMU.

These logical networks can be implemented either on top of a shared physical NoC, or using separate physical links for maximum throughput. For example, the reference configuration we use implements a dedicated distributed cache network for memory with cache-line-wide channels, a narrow mesh for delegations, and a linear word-wide array for bulk distribution, as illustrated in fig. 3.9. This specific network design uses a space filling curve to establish a multi-scale affinity between core addresses and caches, to preserve maximum data locality in bulk creations regardless of the number of cores involved.

State	Update events
Program counters	Thread/bulk creation, branches
Mappings from logical register windows to the register file	Thread/bulk creation
Logical index ranges	Bulk creation
Bulk synchronizers	Bulk creation, bulk synchronization

Table 3.2: Private state maintained by the TMU.

This state is maintained in dedicated hardware structures close to the TMU.

Unit	Description
Thread Creation Unit (TCU)	Responsible for bulk thread creation and logical index distribution
Register Allocation Unit (RAU)	Responsible for dynamic allocation and de-allocation of register ranges in the RF

Table 3.3: Logical sub-units in the TMU.

Summary

○ The overall chip design proposed by the architects at the University of Amsterdam is a dataflow/von Neumann hybrid architecture with hardware multithreading, which uses a large register file as the synchronization name space for split-phase long-latency operations. The core architecture has been co-designed with a dedicated *concurrency management protocol*, implemented in hardware in a coordinating Thread Management Unit (TMU), for controlling hardware threads and composing cores into a many-core chip. We summarize its features in tables 3.1 to 3.3. It is programmed via input *control signals*, also named "concurrency management events." These control signals are in turn generated by the pipeline in response to *new machine instructions*, or from a Network-on-Chip via *active messages*. We describe its programming interface further in chapter 4.
The architecture does not constrain the memory implementation but strongly suggests a scalable memory architecture with support for split-phase transactions and request pipelining. Research is ongoing to provide a distributed cache network to that effect. Remote concurrent work creation and synchronization is supported with hardware primitives for active messages and remote register accesses on the NoC. Bulk creation and synchronization is further optimized by providing low-overhead automatic distribution across multiple cores by the TMU.

Chapter 4

Machine model and hardware interface

Abstract

This chapter introduces the machine model and the machine interfaces offered by various implementations of hardware microthreading. We present the general concepts common to all implementations, then review how design choices may lead to different machines interfaces. We identify major "generations" of interfaces and introduce the low-level assembly languages available to program the various implementations.

Contents

4.1 Overview

This chapter is organized as follows:

- section 4.2 presents the general concepts underlying the machine model, and section 4.3 explores the various semantics that can be attached to these concepts in specific implementations;
- section 4.4 explains how the scheduling hints and thread termination events can be provided by programs;
- section 4.5 reviews how the introduction of hardware microthreading in an existing ISA interacts with the substrate ISA's features;
- section 4.6 outlines potential behaviors for unplanned or erroneous situations;
- section 4.7 introduces actual implementations and their specific interfaces, and recognizes three major "generations" of implementations. Section 4.8 then reviews the concrete assembly languages for these implementations.

4.2 Concepts

○ Regardless of the specific implementation choices, a hardware microthreaded architecture provides the following:

thread contexts, logical threads and thread programs.

Thread contexts are management structures in the Thread Management Unit (TMU) that define individually scheduled instruction streams in the microthreaded pipeline. They are analogous to the "threads" of [II11a, II11b] and existing threading APIs (e.g. POSIX [Ins95]), the "workers" of OpenMP [Ope08] or the "harts" of [PHA10].

Logical threads are the individual units of work defined by programs via the bulk creation event, using a common starting PC and a logical index range. The TMU maps logical threads sequentially over one or more thread contexts. Once activated, logical threads execute a *thread program* from the initial PC to completion. As such, logical threads are akin to the "blocks" of Grand Central Dispatch (GCD) [Sir09, App] or the "tasks" of OpenMP and Chapel [CCZ07].

Thread contexts are independently scheduled, and may interleave while they are active. However, there is no interleaving of logical threads within single thread contexts. Only logical threads are visible to programs; programs can control thread contexts only indirectly via the *placement* of work, discussed below.

thread program actions and synchronization.

Thread programs contain instructions that perform *actions* on the environment. Each instruction has some encoding which specifies its *input and output operands*. The encoding is assumed to derive from a RISC ISA. In particular, operands are encoded either as immediate *values* or as a register *name*, which is a *fixed* offset into a local storage space.

Compared to a conventional register machine, where this storage space would be general-purpose memory cells in hardware, i.e. physical registers, the microthreaded core maps instruction operands to *dataflow synchronizers*. These implement I-variables: a storage cell which may either contain a value ("full") or be "waiting" on a value not yet available [ANP87]. These are then used to provide asynchrony between individual instructions, as a long-latency operation can now simply set its output operand to

"waiting" instead of actually stalling the processor. In other words, the microthreaded machine model subsumes existing instruction sets by substituting dataflow synchronizers for registers. This design was originally proposed in Denelcor's HEP [Smi81] and Tera's MTA [SCB+98] (later Cray's XMT).

However, *memory outside of the core is not synchronizing*: the *indirect* storage accessible via "load" and "store" instructions behaves as regular data cells which always hold a value, i.e. do not synchronize. This is where the design diverges conceptually from the HEP and the MTA, both of which provide synchronization on the entire memory space. The rationale for this stems from the observation that negotiating synchronization is a communication activity, and that in general-purpose environments the software implementer is most competent to recognize computation patterns and organize communication. By restricting implicit synchronization to core-local structures, the machine interface defers the responsibility to organize non-local activities, including arbitrary patterns of inter-core synchronization, to explicit remote synchronizers read/write operations controlled by software.

synchronous vs. asynchronous operations.

The existence of dataflow synchronizers allows us to distinguish between:

- *synchronous* operations, which complete with a full output operand. These guarantee that any further instruction using the same operand as input will not suspend.
- *asynchronous* operations, which complete with a non-full operand in the issuing thread, while letting the operation run concurrently. These can be said to return a "future"[1] on their result.

bounded and unbounded operation latencies.

The existence of asynchronous behavior mandates a comment on operation latency, necessary to determine conditions to *progress* in executing programs.

In the proposed design we can distinguish between:

- operations with *bounded latency*: once the operation is issued, it is guaranteed to complete within a finite amount of time regardless of non-local system activity;
- operations with *unbounded latency*: *whether* the operation completes at all is dependent on non-local system activity, e.g. data-dependent branches in the control flow of other threads.

Intuitively, computation operations issued by programs should have a bounded latency, especially arithmetic and control flow. Yet some unbounded latencies become possible depending on the choice of semantics for concurrency management, discussed in section 4.3 below. In general, memory loads and stores, FPU operations, remote synchronizer reads/writes, and bulk creation *after* allocation of a creation context have a bounded latency; whereas requests to allocate new concurrency resources and requests to synchronize on termination of another thread may be indefinitely delayed by non-terminating threads, and these operations thus have an unbounded latency.

local wait continuations, remote wake ups.

The synchronizing storage is "local" to the core's pipeline and its state is maintained by the flow of local instructions. In particular, *only local instructions can suspend on a*

[1] The concept of "future" is introduced in [Hal85]: *"The construct* `(future X)` *immediately returns a future for the value of the expression X and concurrently begins evaluating X. When the evaluation of X yields a value, that value replaces the future."*

dataflow synchronizer. This choice guarantees that wake-up events to waiting threads are always resolved locally upon writes to synchronizers.

Yet, synchronizers can be accessed remotely across the NoC. For example, a thread running on one core may write remotely to another core's synchronizers and wake up any threads waiting on it on that other core.

virtual mapping of the synchronization space, dataflow channels.

The fixed offsets in instruction operands do not have a static mapping to synchronizing storage; instead, the set of *visible dataflow synchronizers* can be *configured per thread* during bulk creation, and define a "window" on the synchronizing storage. In particular:

- if a synchronizer is visible from only one thread it is said to be "private," or "local," to that thread. Since values stored in it are subsequently readable by further instructions, it behaves functionally like a regular general-purpose register.
- two or more threads can map some of their register names to the same synchronizers. When this occurs, the configuration can be said to implement a *dataflow channel* between the threads: a "consumer" instruction in one thread will synchronize with a "producer" instruction in a different thread. Also, "consumer" instructions which read from a dataflow channel may have an unbounded latency as per the definition above.

bulk creation contexts.

Programs use bulk creation of logical threads to define work. Yet bulk creation is itself a multi-phase *process* which involves e.g. reserving thread contexts in hardware and effecting the creation of logical threads (cf. section 3.3.5). This process itself has both input parameters and internal state. Its parameters are the initial PC, a logical index range, and potentially configuration information for the mapping of register names to synchronizers. Its internal state tracks which logical threads have been created and terminated over time.

Both the parameters and internal state must be kept live *until all logical threads have been created*, especially when some thread contexts must be reused for successive logical threads. As such they constitute collectively a *bulk creation context* which must be allocated, configured and managed.

dataflow synchronizer contexts.

During bulk creation, the TMU can allocate a subset of the synchronizing data storage to map it in the visible window of new threads. Conversely, synchronizers can be released when threads terminate if they are not mapped elsewhere. The set of dataflow synchronizers allocated to a group of bulk created threads thus forms their *dataflow synchronizer context*, a resource that must be managed jointly with the threads.

bulk synchronizers.

Programs use bulk synchronization to wait on termination of threads. As discussed in section 3.3.5 this implies space to store "what to do on termination" for the group of threads waited upon, and a semaphore. These *bulk synchronizers*, separate from the dataflow synchronizers, also constitute state which must be allocated, configured and managed.

automatic work placement and distribution.

When issuing bulk synchronizations, programs can define that the work is to be created either locally or remotely, and on either one or multiple cores. This *placement information* is provided early and serves to route the bulk creation event to the appro-

priate TMU on chip. When targeting multiple cores in one request, multiple TMUs cooperate to spread the logical index range over the cores.
Beyond the creation of threads, the TMUs also assist with the distribution of data. A single event can be sent after bulk creation to broadcast a value to the synchronizers of multiple cores, to serve as input to the logical threads.

binding of logical threads to cores.
In the proposed design, there is no mechanism provided to migrate the contents of management structures across cores after they are allocated. Since the state of synchronizers is not *observable* in software either, this implies that *logical threads cannot be migrated* to another core after they are created.

4.2.1 Families of logical threads

From the previous concepts, we can derive the notion of "*family*" to designate the set of logical threads that are created from a single bulk creation.

Like logical threads, families are defined by programs and *only exist indirectly* through the bulk creation contexts, thread contexts and bulk synchronizers allocated and managed for them in hardware. Although it merely designates an abstract concept, the term "family" forms a useful shorthand for the collective work performed by a bulk created set of logical threads. It is used with this meaning in the remainder of our dissertation and other literature about hardware microthreading.

4.3 Semantics

For the TMU to be *programmable*, some set of *semantics* must be associated to its state structures and their control events. Semantics establish the relationships between state structures and how state evolves at run-time in response to events. To define these semantics, previous research on hardware microthreading has explored three mostly orthogonal aspects:

- how to configure and trigger bulk creation and bulk synchronization (section 4.3.1);
- how to manage the mapping of logical threads to cores and thread contexts (section 4.3.2);
- how to manage the mapping of logical threads to dataflow synchronizers (section 4.3.3).

4.3.1 Bulk creation and synchronization

○ The organization of bulk creation and synchronization determines primarily the relationship between bulk creation contexts, bulk synchronizers and thread contexts. We do not consider here how many thread contexts are allocated and how many logical threads are created over them, this will be covered in section 4.3.2.

Any choice of semantics must respect the following dependencies:

- a bulk creation context must exist before the allocation of thread contexts and the creation of logical threads starts;
- a bulk creation context must persist until all logical threads have been created;
- a bulk synchronizer must exist before:
  - the earliest point at which another thread may issue a request for bulk synchronization;

  - the first logical thread is created;

  whichever comes first;
- a bulk synchronizer must persist until after:
  - the latest point at which another thread may request bulk synchronization;
  - all threads have terminated and a requesting thread is notified;
  - the bulk synchronizer is explicitly released;

  whichever comes last.

Then any choice of semantics must select the *interface* to use in programs, i.e. :

- what machine instruction(s) trigger these processes;
- what parameters can be provided and how;
- under which condition a bulk creation or bulk synchronization request synchronizes with the issuing thread;
- what "output" either of these requests returns to the issuing thread.

In our work, all implementations provide at least the following primitives:

allocation of a bulk creation context.
: This takes as input some placement information, and performs the allocation of *both* a bulk creation context, a bulk synchronizer, a thread context and a set of synchronizers, on the target core(s). It is an asynchronous operation which produces a future on an *identifier to the bulk creation context* to the requesting thread. Its latency is discussed in section 4.3.1.1 below.
The reason why the allocation is combined is to guarantee that once the initial allocation succeeds, it will always be possible to create logical threads and synchronize on their termination afterwards.

configuration of bulk creation.
: This takes as input an identifier to a bulk creation context and writes a value in the corresponding hardware structures, possibly remotely. It is an asynchronous operation with bounded latency and no result.

Implementations then differ in how they deal with allocation failures, and how they trigger bulk creation, synchronization and resource de-allocation. We discuss these further in sections 4.3.1.1 and 4.3.1.2.

4.3.1.1 Allocation failures

○ We have found three main classes of interfaces that differ in how they deal with allocation failures:

- in a *suspending* interface, allocation always suspends until resources become available. In these semantics, allocation always *appears* to succeed from the perspective of running threads but may have an unbounded latency.
- in a *soft failure* interface, allocation failures cause a failure code to be reported as a value to the requesting thread with a bounded latency, for handling by the requesting thread program. This allows the thread program to opt for an alternate strategy, e.g. allocation with different placement parameters or serializing the work.

- in a *trapping* interface, allocation failures are handled within a bounded latency as a fault, and trigger a trap to be handled by a "fault handler" separate from (and possibly invisible to) the thread program.

We found implementations providing both suspending and soft failure interfaces, as discussed in chapter 10. The latter trapping interface is theoretical as of this writing, yet we believe it will become relevant in future work where issues of placement are managed by system software separate from application code.

4.3.1.2 Fused vs. detached creation

○ We found two main classes of implementations that differ in how they trigger bulk creation and synchronization:

- in a *fused creation* interface, bulk creation, bulk synchronization and resource de-allocation are fused in a single asynchronous operation:

  fused creation.
  This takes as input an identifier to a bulk creation context and an initial PC. After issue, the operation triggers the start of logical thread creation, and *also* requests bulk synchronization and resource release on termination of all logical threads. The request for bulk synchronization binds the output operand of the fused creation operation with the bulk synchronizer.
  The operation thus produces a *future on termination* of the logical threads, with an unbounded latency. All the bound resources are de-allocated automatically after completion is signalled.

- in a *detached creation* interface, bulk creation, bulk synchronization and resource de-allocation become independent asynchronous operations:

  creation.
  This takes as input an identifier to a bulk creation context and an initial PC. After issue, the operation triggers the start of extra context allocation and logical thread creation. As soon as logical thread creation starts (which may be later than the issue time, due to network latencies), an identifier for the bulk synchronizer is returned to the issuing thread.
  The creation operation thus produces a *future on the start of creation* with a bounded latency. The latency is bounded because the prior allocation guarantees that creation is always possible.

  synchronization on termination.
  This takes as input an identifier to a bulk synchronizer. After issue, the operation binds its output operand to the named bulk synchronizer.
  It thus returns a *future on termination* of the logical threads with an unbounded latency.

  de-allocation.
  This takes as input an identifier to a bulk creation context. It has no output and completes with a bounded latency. The operation triggers either de-allocation of the resources if the work has completed, or automatic de-allocation on completion if the work has not completed yet.
  This interface does not define the behavior when a synchronization request is issued after a de-allocation request (there is then a race condition between thread

termination and the bulk synchronization request); however, it guarantees that bulk synchronization is signalled before de-allocation if it was registered first.

4.3.2 Mapping of logical threads to cores and contexts

The design proposes to minimize the *necessary* amount of placement information to provide during allocation, while providing extra control to thread programs when desired. The placement information can specify "where" on the chip to allocate in terms of cores, then "how much" on each core to allocate in terms of thread contexts, then "how to spread" the logical index range over the selected thread contexts.

4.3.2.1 Mapping to cores

○ To select the target core(s), the following parameters are available to programs:

inherit. Also called "default" placement in previous work. The placement information that was used to bulk create the issuing thread is reused for the new bulk creation request.
local. The bulk creation uses only the core where the issuing thread is running.
explicit. An explicit parameter is given by the program to target a named core, or cluster, on the chip. The format of explicit placements then depends on the implementation, although recent research has converged to provide a system-independent addressing scheme (cf. chapter 11).

These parameters are intended to control locality and independence of scheduling between application components.

4.3.2.2 Selection of thread contexts

○ To select thread contexts, a single parameter called "block size" or "window size," controls the *maximum number of thread contexts* to allocate *per core* during bulk creation. This parameter is optional; a program may choose to leave it unspecified, in which case the hardware will select a default value. This parameter bounds per-core concurrency, which is useful to control the working set size, and thus cache utilization, on each core. It can also help control utilization of the synchronizer space as explained in [CA88].

In the implementations we have encountered, an undefined block size selects *maximum concurrency*, i.e. allocate all thread contexts available on each core, or the number of logical threads to execute per core, whichever is lower. However, we are also aware of ongoing discussions to change this default to a value that selects the number of thread contexts based on the local cache sizes, to minimize contention. The choice of default behavior should thus be considered implementation-dependent.

4.3.2.3 Distribution of the logical indexes

○ In the proposed interface, a requesting thread can specify an index range by means of *start*, *limit* and *step* parameters (side note 4.1). Compared to a potentially simpler interface which would only allow programs to set the number of logical threads, this interface enables direct iteration over data sets, in particular the addresses of array items in memory. These parameters are optional, and default to 0, 1, 1, respectively.

Side note 4.1: Logical index sequence.

The logical thread index sequence is defined from the *start*, *limit*, *step* parameters by using a counter initialized with *start* and increased with *step* increments until it reaches *limit*. *limit* is excluded from the sequence.

In the implementations we have encountered, no further control is provided to programs over the distribution of logical threads. The index space is spread evenly over all selected cores. On each core, thread contexts "grab" logical thread indexes on a first-come, first-served basis until the local index pool is exhausted. Again, we are aware of ongoing discussions to provide more control to programs to facilitate load balancing, a topic which we revisit in chapter 13. The interfaces available to programs to control logical distribution should thus be considered an implementation-dependent design choice, orthogonal to the general concepts of hardware microthreading.

In each logical thread created, the first synchronizer is pre-populated by the TMU with the value of the logical index.

4.3.3 Mapping of threads to synchronizers

As explained in section 3.3.3, the proposed architecture uses a dynamic mapping between operand addresses in instructions and the synchronization space. *How many* synchronizers are visible to each thread, and *which* synchronizers are visible, are both configurable properties of the thread context, fed into the pipeline for reading instruction inputs when each instruction is scheduled.

4.3.3.1 Interface design requirements

○ The question then comes how to design an interface to configure this mapping. Here two forces oppose: on the one hand, maximum flexibility for software would mandate the opportunity for arbitrary mappings; on the other hand, the complexity of the mapping parameters impacts logic and energy costs in the hardware implementation.

Any trade-off when designing an interface must consider the following:

- for the design to be truly general-purpose, the opportunity must be given to programs to specify at least two private (non-shared) synchronizers per thread. This is the theoretical minimum necessary to carry out arithmetic and recursion within a thread;
- any configuration that shares synchronizers between threads should establish clear conditions for the eventual release of the synchronizers, lest their management becomes a source of tremendous complexity in compilers;
- a key feature of the design is the short latency bulk creation of logical threads over thread contexts. Any design which requires the hardware to modify the mapping of synchronizers between logical threads, or maintain heterogeneous mapping information across thread contexts, will impact the latency of thread creation.

Another aspect is the selection of *where the configuration is specified*. When considering only performance and implementation costs, the cheapest and most efficient choice is to let the thread that issues a bulk creation provide the mapping parameters explicitly. Yet this choice would be shortsighted. From a programmability perspective, a clear advantage of the architecture is the ability to compose separately written thread programs via bulk

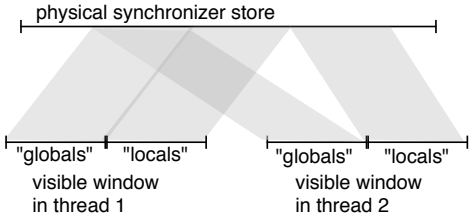


Figure 4.1: Example window mapping for two sibling threads with global and local synchronizers.

creation. The knowledge of how many synchronizers are visible, and whether and how they are shared between threads, is needed to assemble instruction codes. It thus belongs with the code generator for the threads being created, not the threads doing the creation. Therefore, an implementation should associate this configuration information with the thread programs that are the target of bulk creation somehow.

4.3.3.2 Common semantics

○ We could find the following consensus throughout implementations:

- *The configuration information immediately precedes the first instruction of a thread program.* This answers the argument above, and provides good locality of access to the creation process.
- *Thread programs can specify a number of private synchronizers.* This answers the generality argument above. The fixed number is then allocated for each participating thread context and is reused by all successive logical threads running on these contexts. These synchronizers are dubbed "*local*" in the remainder of our dissertation.
- *Thread programs can specify a number of synchronizers to become visible from all threads.* This is meant to provide common data to all threads. That fixed number of synchronizers is mapped onto the visible window of all participating thread contexts on that core. These synchronizers are henceforth dubbed "*globals*"; they are "global" to all logical threads created from a single bulk creation. We also call them "global dataflow channels" when they are used strictly for broadcasting.

An example is given in fig. 4.1.

4.3.3.3 Heterogeneous sharing patterns

○ Besides the "global" pattern introduced above, prior work has explored other forms of synchronizer sharing between threads. The implementations we have used have proposed the following features, in various combinations:

"shared" synchronizers between adjacent thread contexts.
With this feature, thread programs can specify a number of extra synchronizers to share between neighbouring contexts during bulk creation. This creates the overlap pattern illustrated in fig. 4.2. This feature is coordinated by the TMU so that logical thread indexes are distributed accordingly, and a point-to-point communication chain

Side note 4.2: Purpose and motivation of "shared" synchronizers.

This feature was originally proposed to explore whether bulk created microthreads are a suitable alternative to dependent sequential loops, when the loop-carried dependency has stride 1. We revisit this in section 13.8.

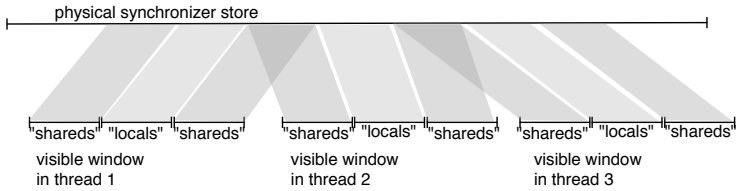
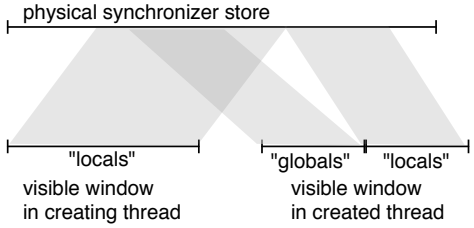


Figure 4.2: Example window mapping for three sibling threads with local and shared synchronizers.

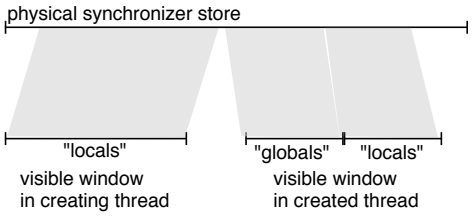
is created in the logical thread order. The mapping is heterogeneous because the "border" thread contexts do not necessarily share synchronizers (cf. below).
When this pattern is used only to express a forward-only dependency chain, we call the synchronizers "shared dataflow channels." When necessary, from the perspective of a given thread we distinguish further between "D" synchronizers which are shared with the preceding context, and "S" synchronizers shared with the succeeding context.

"hanging" vs. "separated" global synchronizers.
In the "separated" variant, the global synchronizers of new threads are freshly allocated from the synchronizing storage and initialized to the "empty" state upon bulk creation, on each participating core. A dedicated NoC message is then available to programs to broadcast a value to the participating cores explicitly.
In the "hanging" variant, if the core where the thread issuing a bulk creation is also a target of the creation, then on that core the global synchronizers of new threads are not freshly allocated; instead, the window of the new threads is configured to map to some *existing* private synchronizers of the thread that issued the bulk creation. Here, the interface allows a creating thread to indicate which of its local synchronizers to use as a base offset for the created windows.
Then, if there are more cores participating, fresh synchronizers are allocated on the remaining cores. The values stored in the synchronizers on the first core are automatically broadcasted to the other cores during bulk creation.
The "hanging" variant results in higher utilization of the synchronizing store on the first core. It was historically the first implemented. We can see it was primarily designed for single-core systems and creates a strong resource dependency between the creating thread and the created threads. The "separated" variant is more flexible and homogeneous when multiple cores are involved.
We illustrate the difference between "hanging" globals and the "separated" alternative in fig. 4.3.

"hanging" vs. "separated" shared synchronizers.
As with "separated" globals above, with "separated" shareds the shared synchronizers of new threads are freshly allocated and initialized to "empty." The "leftmost" synchronizers in the first thread context may then be subsequently written asynchronously by the issuing thread using a dedicated NoC message.

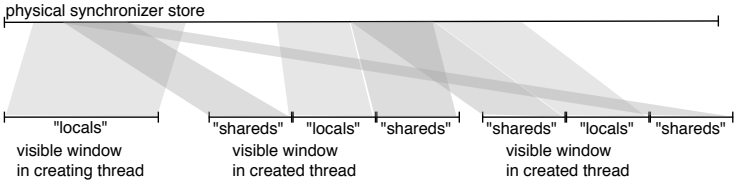


(a) Mapping of "hanging" globals. The local synchronizers of the creating thread are shared with the created threads.

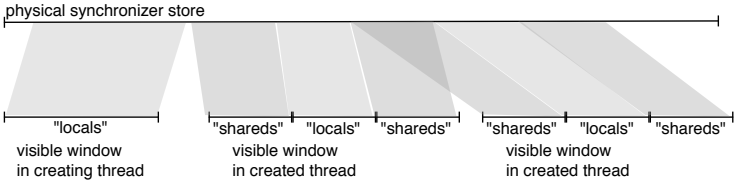


(b) Mapping of "separated" globals. The created threads uses independent registers.

Figure 4.3: Alternatives for mapping global synchronizers.



(a) Mapping of "hanging" shareds. The local synchronizers of the creating thread are shared with the created threads.



(b) Mapping of "separated" shareds. The created threads uses independent registers.

Figure 4.4: Alternatives for mapping shared synchronizers.

With "hanging" shareds, as with "hanging globals" above, a special case exists if the core where a bulk creation is issued from is also participating. In this case the shared synchronizers of the border contexts are not freshly allocated from the synchronizing storage; instead, their visible windows are configured to map to some existing private synchronizers of the issuing thread.
As above, the "hanging" arrangement was the first designed, with single-core execution in mind. We illustrate the difference between "hanging" shareds and the "separated" alternative in fig. 4.4.

4.3.3.4 Layout of the visible window

○ Once a program has determined *which* synchronizers are visible, a choice exists of *where* in the visible window they should be mapped. For example, if a thread program configures e.g. 4 local synchronizers and 7 globals, it is possible to map offsets 0 to 3 to the locals, and offsets 4 to 10 to the globals. It is also possible to map offsets 0 to 6 to the globals, and offsets 7 to 10 to the locals.

We can identify this *mapping order*, using a string of the form "X-Y-Z...", where X, Y, Z... indicate the type of synchronizer mapped and the order in the sequence indicates the order of mapping in the visible window. For example, some early implementations used the order G-D-L-S, meaning that global synchronizers were mapped first (if any), followed by the synchronizers shared with the previous context, followed by the local synchronizers, followed by the synchronizers shared with the next context.

□ From the hardware designer's perspective, this choice seems neutral. However, here we discovered an extra requirement from the programmability perspective. Indeed, considering a common software service that uses only local synchronizers (say, a routine to allocate heap memory), it proves useful to be able to reuse the same code for this service from multiple thread contexts with different numbers of "global" and "shared" synchronizers. However, the address of synchronizers are statically encoded in the instruction codes; this implies that *the mapping of local synchronizers must be identical regardless of the number of other synchronizer types.* To satisfy this, we have required that local synchronizers are always mapped first in the visible windows of threads. Following our requirement the implementations have been subsequently aligned to the order L-G-S-D.

4.4 Out of band control bits

○ Introduced in section 3.2.1, the instruction *control bits* indicate to the fetch unit when to *switch* threads and when to *terminate execution* of a thread. These bits must be defined for every instruction processed through the pipeline; there are three possible design directions to program them.

The direction least intrusive for existing software, that is, *able to preserve an existing code layout from a pre-existing ISA*, is also the most expensive in hardware: maintain two PCs on the fetch unit, where one refers to code and the other to the control bits in a separate area in memory. In this approach, the I-cache is shadowed by a cache for control bits, or "C-cache." Thread creation events should specify both the initial PC for instruction and for control bits. Alternatively, the TMU can read the control bit PC by looking up thread metadata at a fixed offset from the initial instruction PC. The fetch unit must further place threads on the waiting list on either an I-cache miss, or a C-cache miss. The extra complexity of this approach comes from the synchronization of waiting threads: upon I-cache refills, a thread can only migrate from the "waiting" state to "active" if the C-cache hits, and *vice-versa*.

The most contrasting approach is to make the control bits part of the machine instruction encoding, for example using two leading bits in the instruction format. With variably sized instruction words, an existing ISA can be extended this way, however with any fixed-width ISA (such as found in RISC designs) this approach requires to redesign the ISA because some existing instructions may have no unused bits left. In the latter case, this approach is also the most intrusive as it prevents reusing existing assembly code and compilers as-is.

The implementations we have encountered opt instead for an intermediate approach: interleave control bits and machine instructions. In this approach, each group of N instructions is immediately preceded by $2N$ control bits. To ensure that the fetch unit can always read the control bits, these must be present in the same I-cache line as the controlled instructions. This in turn constrains the minimum I-cache line size. With 32-bit instruction formats, the "sweet spot" which incurs no wastage of space in I-lines while preserving a size power of two lies at $N = 15$, with a minimum I-cache line size of 64 bytes. We detail the optimal parameters in Appendix B.

Side note 4.3: Thread switching not specified by control bits.

Besides where specified by control bits, thread switches are also incurred automatically for instructions laying on the boundary of I-cache lines. This avoids missing the I-cache during fetch (side note 3.3), but also helps guarantee fairness of scheduling between active threads. Switches are also incurred upon branches for fairness. Yet, for efficiency it is recommended to still place a "switch" annotation on the branch instruction, as this will ensure the next pipeline slot is filled by an instruction from another thread and avoids a one cycle pipeline bubble if the branch is taken.

Side note 4.4: Switching and thread termination as instructions.

Another approach may consider using explicit instructions for thread switching and termination. An instruction can signal a control event prior to the instruction that immediately succeeds it. This requires embedding the few gates necessary to decode the instruction within the fetch stage itself. Alternatively, an instruction can signal the event for the end of the next cycle. In this case, another instruction that immediately follows would still be executed before the event is effected.
In either case, this solution incurs a pipeline bubble at every control event: the instruction occupies an idle cycle that does not contribute to useful computations. It is thus only interesting if switching and thread termination are uncommon. This solution is thus undesirable with fine-grained interleaving, e.g. where threads replace inner loop bodies and where memory loads incur switching, such as with the proposed design.

To program this latter approach, the assembler program must emit a control word at regular intervals between instructions. This is covered below in section 4.8.

4.5 Interactions with a substrate ISA

The proposed architecture concepts can be applied either to new RISC cores with a dedicated ISA, or *extend* existing RISC cores, and thus to pre-existing ISAs. In the latter case, the ISA is a *substrate* upon which the extensions are built. When working with a pre-existing ISA, its semantics interact with the proposed micro-architecture. We list the most notable possible interactions below.

4.5.1 Delayed branches

○ Delayed branches, as found in the MIPS, SPARC and OpenRISC[2] ISAs, have been designed for in-order, single-threaded pipelines to reduce the cost of control hazards: the instruction(s) immediately succeeding a branch execute(s) regardless of whether the branch is taken; the corresponding positions in the instruction stream are called *delay slots*. With "branch and link" instructions, where the PC of the first instruction after the branch is saved to another register, the first address *after* the delay slot(s) is used.

When an architecture using delay slots evolves to use out-of-order execution, e.g. via the introduction of superscalar execution or pipeline multithreading, the "next instruction" in instruction stream order may not immediately succeed the branch in the pipeline anymore, and extra logic must be introduced to ensure that this instruction completes even when the branch is taken. With hardware multithreading, this implies maintaining a separate "next PC" register for every thread.

[2] `http://opencores.org/or1k`

4.5.2 Predication

○ Predication, as found in the ARM and IA-64 ISAs, is another feature designed to reduce the cost of control hazards: each instruction is predicated on shared *condition codes* updated by special instructions.

When an architecture using predication evolves to use out-of-order execution, extra logic must be introduced to ensure that the ordering of tests to the condition codes follow the ordering of updates. With hardware multithreading, this further implies that separate condition codes must be maintained for each thread.

4.5.3 Register classes

○ Some ISAs rely on *separate register classes* for the operands and targets of certain instructions. Most processor designs use different register classes for integer and floating-point instructions; then ARM also has a separate class for "return addresses," used implicitly by branch and link instructions, and SPARC has "status registers," including the Y register used as implicit operand for multiplies and divides.

With hardware multithreading, separate instances of these registers must be available in each thread. Moreover, if these registers can become the target of long-latency operations, such as the Y register in SPARC, the ordering of dependent instructions cannot be controlled by the dataflow scheduling from section 3.2.1 unless these registers are also equipped with dataflow state bits.

4.5.4 Multiple register windows per thread

○ We have outlined the role of sliding register windows in section 3.3.3. More generally, some ISAs define multiple register windows *per thread* and conditions to "switch" the instruction stream from one to another. For example, ARM defines that a separate register window is used upon traps, syscalls and interrupts.

With the introduction of hardware multithreading, different instances of the register windows must be available to each thread to avoid saving and restoring the registers to and from memory during context switches. The number of windows *per thread* does not change as the desired number of hardware threads increases, and becomes the growth factor for the register file. However, mere duplication would be wasteful, as only one window is active per instruction and per thread. Here two optimizations are possible.

When the ISA specifies that extra register windows are used only for asynchronous events, such as in ARM, it is possible to serialize all the asynchronous events from all threads over a smaller set of registers. In this scenario, if a trap handler is ongoing for one thread, a trap from another thread would suspend until the trap register window becomes available. While this approach may suggest reduced parallelism, it is quite appropriate for trap and syscall entry points which usually require atomic access to the core's resources (e.g. to fetch exception statuses).

If the ISA defines explicit instructions to switch register windows, it is possible to replace these instructions by explicit loads and stores to exchange the registers to memory. This can be done either statically by substitution in a compiler (as we did, cf. Appendix H.9), or dynamically by a dedicated process in hardware. This trades the complexity of a larger register file for overhead in switching windows within threads, which can be tolerated using dataflow scheduling (section 3.2.2).

ISA	MIPS	SPARC	Alpha	ARM	PowerPC	OpenRISC
Delay slots	yes	yes	no	no	no	yes
Predication	no	no	no	yes	no	no
Register classes	2	3	2	3	2-6	2
Status register as implicit operand	no	yes	no	yes	yes	yes
Register windows	no	yes (8+)	no	yes (6)	no	no
Max. operands per instruction (input/output)	2/1	4/2[†]	2/1	3/2[‡]	2/1	2/1

[†] `std` has 4 inputs and 1 output; `ldd` has 2 inputs and 2 outputs.
[‡] dual multiplies have 3 inputs; long multiplies have 2 outputs.

Table 4.1: Features of existing general-purpose RISC ISAs.

4.5.5 Atomic transactions

○ With RISC-style split-phase atomic transactions, for example the Load-link, Store-conditional (LL/SC) instruction pairs found in MIPS, PowerPC, Alpha and ARM, the semantics can be kept unchanged when the core is extended with multithreading, with no extra complexity. However, the rate of transaction rollbacks may increase significantly if the working set of all local threads does not fit in the L1 cache.

In contrast, if the ISA features single instruction atomics, in particular Compare-and-Swap (CAS) or atomic fetch-and-add found in most other architectures than the four already named, the memory interface must be extended with a dedicated asynchronous FU for atomics. Indeed, single-instruction atomics require locking the L1 line for the duration of the operation, and without an asynchronous FU, any subsequent load would cause the pipeline to stall until the CAS instruction is resolved.

4.5.6 Summary of the ISA interactions

This section has outlined the interactions between features of the substrate ISA and the introduction of hardware microthreading. Of common pre-existing general-purpose ISAs (table 4.1), the Alpha ISA is the one that requires the least added complexity when introducing microthreading. In our research, we have used implementations derived from both the Alpha and SPARC substrate ISAs:

- The original Alpha ISA uses neither delay slots, nor predication, nor sliding register windows, and uses only two symmetric register classes for integer and floating-point instructions, with no implicit operands in instructions and only one window per thread. It does not define CAS instructions but does feature LL/SC.
- The original SPARC ISA uses delay slots and sliding register windows, but no predication. It has three register classes, with implicit operands in integer multiply and divide (the Y register). It also defines CAS instructions, and 8 sliding windows per thread.

4.6 Faults and undefined behavior

○ To increase understanding of the proposed interface, we found it useful to explore the "negative space" of the semantics. That is, provide information about situations that "should not occur" but still may occur due to programming errors, malicious code, etc. From the hardware interface perspective, we can *specify* two types of reactions upon an erroneous or unplanned situation in hardware.

If we specify that a situation constitutes a *fault*, we mean that it constitutes an error recognized by the implementation, and where execution is guaranteed to not progress past the fault. For example, after a fault occurs, either the entire system stops execution, or a *fault handler* is activated to address the situation and decide an alternate behavior.

If we specify *undefined behavior*, we mean a situation where the behavior of an implementation is left undefined, that is, an implementation may or may not test for the situation and execution may or may not progress past the situation. The characteristic of undefined behavior is *absence of knowledge about the behavior*, in particular that *no further knowledge about the state of the system can be derived from past knowledge after the situation occurs.*

We detail the situations we have found most relevant to programmability and validation in the following sections.

4.6.1 Invalid synchronizer accesses

4.6.1.1 Unmapped synchronizers

If an instruction uses a synchronizer address that is not mapped in the visible window (e.g. address "8" when there are only 4 synchronizers mapped), two behaviors are possible:

- either an implementation guarantees that such situations will always read the value zero as if it was a valid literal operand, or
- the situation constitutes a fault.

We found implementations exhibiting both these behaviors.

4.6.1.2 Stale states and values in local synchronizers

○ Upon bulk creation, "local" synchronizers are freshly allocated for each thread context (section 4.3.3.2). Yet an implementation may choose to not reset the dataflow state after allocation. If that is the case, the synchronizer stays in the state it was from a prior use by another group of threads. It may contain a value, or it may be in the "empty" state. Therefore, in a new thread, if an instruction reads from a local synchronizer *before it has been written to* by another instruction from the same thread, two behaviors are possible:

- either the instruction reads from the previous state unchanged, and may read a stale value or deadlocks the thread if the operand was in the "empty" state; or
- the implementation guarantees that the value zero is read from any "empty" local synchronizers.

This aspect is crucially important when generating code. Indeed, if the thread program transfers control to a subroutine defined separately, and that subroutine subsequently spills some callee-save synchronizers to memory, this may cause the thread to suspend (and deadlock) if the synchronizers were still in the "empty" state. To avoid this situation, a code

generation would thus need to explicitly clear all local synchronizers to a "full" state prior to calling any subroutine. With the alternate implementation that reads zero from "empty" local synchronizers, this initialization is not necessary.

4.6.1.3 Writing to a non-full synchronizer

○ If an instruction is issued with an output operand containing a future on the result of a previous asynchronous operation from the same thread (e.g. the target of a prior memory load, or the future of a family termination due to bulk synchronization), the following behaviors are possible:

- undefined behavior: the hardware may let the instruction execute and overwrite the continuation of the pending result with a new value; when this happens, any threads that were waiting on that result will never be rescheduled, and a fault may or may not occur when the operation completes and the asynchronous response handler does not find a continuation in the synchronizer; or
- the instruction suspends until the target becomes full and can be written to.

Undefined behavior can be avoided within procedure bodies when compiling programs with clear use-def relationships (as is the case in C), but is more insidious between procedure calls. Indeed, when transferring control between subroutines, a subroutine may spill some callee-save synchronizers to memory, and restore them from memory before returning control. This means that these subroutines end by issuing some memory loads but not waiting on the result. When the caller resumes, it may then choose to not reuse the previous value of the callee-save synchronizer, and instead write a new result to it. Without support from the hardware (second alternative above), a code generator must ensure that all memory loads at the end of a procedure are eventually waited upon, by introducing dummy "drain" instructions.

4.6.2 Invalid bulk creation parameters

○ An invalid program counter during bulk creation (e.g. to non-readable memory) triggers the same behavior as a control flow branch to an invalid address, whatever this behavior is specified to be in the substrate ISA.

Concurrency management faults are then signalled in the following situations:

- the control word (section 4.4) contains invalid bit values;
- a concurrency management operation is provided an input that is not of the right *type*, e.g. the parameter to a create operation is not the identifier to a bulk creation context;
- the *order* of operations in the protocol is violated, e.g. the bulk creation request is sent twice on the same bulk creation context.

A more insidious situation comes from data-dependent behavior in bulk creation. In particular, if an invalid index range is provided, for example with the start value greater than the limit, then this yields undefined behavior in all the implementations we have encountered. Some let the bulk creation process start creating logical threads and never terminate, some others terminate bulk creation when the largest index value is reached. These ought to be signalled as faults, yet we should recognize that detecting the condition would add extra steps in the critical path of bulk creation, increasing its latency.

4.6.3 Deadlock detection and handling

○ With the proposed primitives, deadlocks should only occur as a result of thread programs using the primitives explicitly in an order that creates a deadlock situation, or from the exhaustion of available concurrency resources. For example, a thread may create another logical thread which reads from an input dataflow channel, and then wait on termination of that logical thread without providing a source value on the channel. While these situations are avoidable "by contract" with a programmer or a higher-level code generator, they may nonetheless occur in a running system.

The result is one or more thread contexts that are suspended and never resume execution. The deadlock can then propagate (other threads waiting on termination recursively) or stay isolated. In either case the allocated thread contexts are never released and cannot be reused. This constitute leakage of hardware resources.

The implementations we have encountered do not provision any support for detecting and handling resource leakage due to deadlocks when it occurs. At best, a system-level deadlock detection may be available, which signals to an external operator that the entire system is idle yet there are some threads suspended.

In [JPvT08], the authors propose that resources (core clusters, communication channels) are *leased* to program components for contractual amounts of time or maximum memory capacities. If the contract is violated (e.g. due to a timeout or quota excess), the resource is forcibly reclaimed by the system. The existence of such mechanisms would provide a way to address the deadlock situations mentioned above; however they are not (yet) available in the implementations we worked with.

4.7 Specific implementations and their interfaces

○ During our work we have been exposed to multiple implementations with different design choices for the various aspects covered above. Some of these implementations are listed in table 4.2. We can classify these implementations mostly into six categories along two axes:

1st, 2nd and 3rd generation ISAs. The first generation corresponds to the characteristics captured in the UTLEON3 implementation on FPGA, eventually published in [DKKS10, DKK+12]. The second generation introduces detached creation (cf. section 4.3.1.2), removes "hanging" mappings of synchronizers (cf. section 4.3.3.3) and thus makes new threads fully independent from creating threads; this simplifies code generation as we discuss later in sections 6.3.4 and 6.3.5. The third generation codifies placement across arbitrary cluster of cores, we discuss this further in chapter 11.

SPARCv8 (32-bit) vs. Alpha (64-bit). The SPARC-based implementations extend the SPARCv8 ISA with microthreading, but do not include support for traps, delay slots, sliding register windows, dataflow scheduling on the status registers (including Y), nor CAS. Sliding windows are replaced by explicit spills and restores (cf. Appendix H.9). The Alpha-based implementations extend the Alpha 21264 ISA with microthreading, but do not include support for traps, PALcodes nor LL/SC.

We illustrate these six major implementations in fig. 4.5. The figure also shows how the assembler and linker have evolved with the ISA generations.

Name	Substrate ISA	Allocation failure modes	Creation style	Placement style	Mapping style for "globals"	Mapping style for "shareds"	Window layout
MGSim v1 200807	Alpha	suspend only	fused	local, explicit (local cluster)	hanging	hanging	G-S-L-D
MGSim v1 200902	Alpha, SPARCv8	suspend only	fused	local, explicit (local cluster)	hanging	hanging	G-S-L-D
MGSim v1 200904	Alpha, SPARCv8	suspend only	fused	inherit, local, explicit (named static clusters)	hanging	hanging	G-S-L-D
MGSim v1 200909	Alpha, SPARCv8	suspend, soft fail	fused	inherit, local, explicit (named static clusters)	hanging	hanging	G-S-L-D
MGSim v2 201004	Alpha, SPARCv8	suspend, soft fail	detached	inherit, local, explicit (named static clusters)	separated	separated	G-S-L-D
MGSim v2 201005	Alpha, SPARCv8	suspend, soft fail	detached	inherit, local, explicit (named static clusters)	separated	separated	L-G-S-D
MGSim v3 201103	Alpha, SPARCv8	suspend, soft fail	detached	inherit, local, explicit (arbitrary clusters)	separated	separated	L-G-S-D
UTLEON3 <201003	SPARCv8	suspend only	fused	inherit only	hanging	hanging	G-S-L-D
UTLEON3 201003	SPARCv8	soft fail only	fused	inherit only	hanging	hanging	G-S-L-D
UTLEON3 201011	SPARCv8	soft fail only	fused	inherit only	hanging	hanging	L-G-S-D
utc-ptl	N/A	trapping	detached	local, explicit (local cluster)	hanging	hanging	N/A
dutc-ptl	N/A	trapping	detached	local, explicit (nodes on network)	hanging	hanging	N/A
hlsim <201107	N/A	trapping	detached	local, explicit (local cluster)	hanging	hanging	N/A
hlsim 201107	N/A	suspend, soft fail	detached	inherit, local, explicit (arbitrary clusters)	separated	separated	N/A

Table 4.2: Characteristics of various implementations.

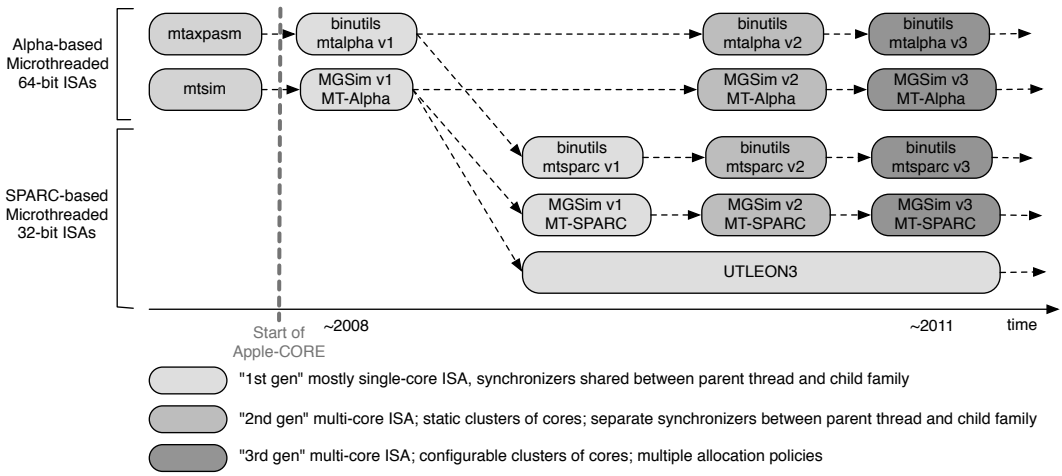


Figure 4.5: History of the hardware platform implementations.

4.8 Assembler and assembly language

The assembler program translates textual machine code into binary code suitable for execution. Any given assembler, and more importantly its *assembly language*, are thus dependent on the specific choice of instruction encoding and the design choices described earlier.

During our work we have been exposed to multiple assemblers, developed outside of our own research. They were all derived from the GNU assemblers for the GNU/Linux/Alpha (`alpha-linux-gnu`) and GNU/Linux/SPARC (`sparc-linux-gnu`) targets.

4.8.1 Common features

○ The following features exist across all implementations:

- the new mnemonics "`swch`" and "`end`" are recognized. The assembler computes the control bits (section 4.4) by associating the value "switch during fetch" to instructions immediately followed by `swch`, and the value "switch in fetch and end thread" to instructions immediately followed by `end`.
- the new directive "`.registers`" is recognized. This should be used at the start of a thread program and produces the synchronizer mapping configuration discussed in section 4.3.3. The numerical arguments specify the number of local synchronizers per thread, the number of "shared" synchronizers shared between adjacent contexts, and the number of "global" synchronizers shared by all contexts. There are different arguments for the different register classes, for example integer and FP registers on Alpha.
- the format of *register names* is extended to provide relatively numbered aliases to the different types of synchronizers in the visible window. For example, the name "`$l3`" on the Alpha on an implementation with a G-S-L-D layout (section 4.3.3.4) is translated by the assembler to the operand offset $G + S + 3$, where G and S are the number of "global" and "shared" synchronizers declared earlier with "`.registers`".
  The formats `$l`N, `$g`N, `$s`N, `$d`N, `$lf`N, `$gf`N, `$sf`N, `$df`N (Alpha), `%tl`N, `%tg`N, `%ts`N, `%td`N, `%tlf`N, `%tgf`N, `%tsf`N, `%tdf`N (SPARC) are recognized: "l" stands for "local"; "g" for "global", and "s" and "d" for the "S" and "D" parts of "shared" synchronizer mappings (section 4.3.3.3). The SPARC variant uses a "`%t`" prefix for aliases to avoid ambiguity with the native SPARC name "`%g0`."

4.8.2 Extra instructions

○ The instruction mnemonics available depend on the choice of interface for bulk creation and synchronization (section 4.3.1). In all implementations we can find the following:

- "`allocate`," which expands to instructions that allocate a bulk creation context. There are different forms depending on the supported failure modes (section 4.3.1.1) and placement options (section 4.3.2.1).
- "`setstart`," "`setlimit`," "`setstep`," "`setblock`." These expands to instruction that configure the parameters of bulk creation (sections 4.3.2.2 and 4.3.2.3).

Further instructions depend on:

- the creation style, discussed in section 4.3.1.2:

  - interfaces with fused creation provide a combined "`create`." (The Alpha ISA further distinguishes between "`cred`" and "`crei`" for immediate/direct and relative/indirect PCs specifications).
  - interfaces with detached creation distinguish between "`create`" for bulk creation, "`sync`" to request bulk synchronization, and "`release`" to request de-allocation of the resources.

- whether synchronizers are "hanging," discussed in section 4.3.3.3:
  - interfaces with "hanging" synchronizers must specify at which offset of the visible window of the issuing thread the sharing of hanging synchronizers should start. Here there are two variants; either there is no interface and the hanging always starts at the first "local" synchronizer (e.g. on UTLEON3); or an instruction "`setregs`" exists to explicitly configure the offset (e.g. on MGSim v1).
  - for interfaces without "hanging" synchronizers, explicit "`put`" and "`get`" instructions are available to communicate values between an issuing thread and the created threads.

□ We have constructed a detailed specification of the exact formats and encodings of the various instructions on the Alpha and SPARC ISA variants of "MGSim v3" and "UTLEON3 201111," reproduced in Appendix D.

4.8.3 Program images

○ The implementations use either flat memory images or ELF [Com95] images to load program code and data into memory. The GNU linker has been modified prior to our work to produce these from object code created by the assemblers introduced above.

Summary

○□ The machine interface of the proposed architecture subsumes an existing ISA by substituting general-purpose registers with dataflow synchronizers. In addition to this, new primitives and extra semantics are added to the existing ISA to control the Thread Management Unit (TMU) in hardware and thus provide control over concurrency management to programs. To illustrate, we provide in Appendix C an example concrete program for a specific implementation.

○ Due to a diversity of possible implementation choices, the specific machine interfaces of given implementations may differ slightly in their semantics. For example a fundamental distinction, visible to programs, exists between "fused" and "detached" bulk creation. We have highlighted the different areas where implementation choices impact the machine interface, and characterized the available implementations by their specific choices in these areas.

Part II

Foreground contribution

—Demonstrating and documenting the general-purpose programmability of a new chip

Chapter 5

System perspective
—Identifying audiences, their expectations and opportunities

> Give a man a fish and you feed him for a day. Teach a man to fish and you feed him for a lifetime. Unless he doesn't like sushi–then you also have to teach him to cook.
>
> Auren Hoffman, cited in [Dac06]

Abstract

The responsibility of the hardware architect extends beyond the design of hardware components on a processor chip, to include meeting the platform expectations of software ecosystems. The reward for entering a dialogue with the audience is the opportunity to gain early feedback upon the innovation. In this chapter, we identify candidate ecosystems for the proposed design from part I, the parties involved in the dialogue and their expectations. This identification allows us to sketch a technical path, which we will then follow throughout part II.

Contents

5.1 Introduction

Ever since the advent of hardware specification languages like Verilog or VHDL, innovators in hardware architectures have been able to separate the design of new *parameterized components* from the selection of actual parameter values.

This raises the question of how to evaluate a new design. In principle, it is possible to look at a parameterized specification in isolation and analyse the impact of design choices locally. For example, the authors of [BJM96] study analytically the optimal number of threads that must be active simultaneously in a microthreaded core to tolerate memory latencies; this number of threads is then itself parameterized by the actual memory latencies in a concrete system. However, local study of a new component is only possible by making assumptions about the context where the component will be used. For example, simulating a new branch predictor using precomputed execution traces assumes that the sequence of instructions on a processor without the branch predictor is the same as when the branch predictor is available. In general, simulations of a new component in isolation are only valid if the models of the component's environment, which serve as input to the simulation, take into account the impact of the innovation on the entire system.

The question of evaluation thus becomes non-trivial when a new component functionally impacts multiple levels of a computing system. Impact on function implies that the structure of software may become different, and therefore that models of system behavior constructed without the component become inaccurate as input to predict the behavior of a system equipped with the new component. In this case, *the construction of a system*, or at least a full system simulation, becomes unavoidable to evaluate the innovation.

This in turn raises the question of how much of a system one should construct to expose and evaluate the innovation. Processor chip designers typically consider that innovation in processor components is fully realized once the processor chip is fully configured, either on silicon or in simulations. Yet the impact of an innovation at the level of processors can only be studied in the context of software workloads. These workloads in turn are highly dependent on the environment of the processor, including memory, of course, but also:

- the logic that define *I/O interfaces*, e.g. an Ethernet adapter, which defines the protocol used by software to react to external asynchronous events;
- the first level *code generation* tools that provide a programming environment sufficient to implement software operating systems, since the operational semantics of the first level programming languages will constrain those of any higher-level language implemented on top of them.

There are thus two ways to expose a new component in processors, depending on the impact of the innovation on the software ecosystem. If existing I/O infrastructure and abstractions, as well as first level programming interfaces, can be reused without significant changes, then it becomes possible to argue that the new design can be used as a "drop-in replacement" in existing systems, i.e. that it "preserves backward compatibility." It then becomes possible to use existing software and evaluation methods soundly, at a low cost. In contrast, if the new design requires changes to any of these aspects, then the party in charge of the innovation must take the responsibility to either implement the corresponding new I/O infrastructure or code generation technology, or partner with third-parties to provide them, before describing how software workloads must be adapted to the new proposed platform.

This is the topic introduced in this chapter, and covered in the rest of part II: we propose a methodology to support the exposition and realization of an innovation in processor chips, when the innovation has a potential impact on the system interface to software.

In section 5.2, we expose the concept of *computing ecosystem* and we highlight the interactions between ecosystem actors in the design of computing platforms. Then in section 5.3, we outline our methodology where the *expectations and requirements* of existing computing ecosystems are used as a starting point for integration. Our key idea is *conceptual backward compatibility*, i.e. the match of existing expectations in new integrations. This idea stems from the observation that hardware and software components are nowadays modular, and that a new platform provider can co-redesign *part of* the hardware platform and the software platform to tune it to a new chip design while reusing most components of existing systems. The benefit of this approach is that any form of partial technology reuse can provide *early feedback* on the substance of the innovation. Our methodology involves first selecting a computing ecosystem, then understanding their socio-technical dynamics, and in particular identifying *the subset of their technology that interfaces software with hardware*. Then we propose to use this understanding to derive integration steps where only this technology subset is replaced by functionally equivalent hardware and software components. We then apply this methodology in section 5.4 to the innovation from part I and derive possible integration steps. We outline in section 5.5 the resulting system platform, which was used subsequently for software developments.

5.2 Computing ecosystems

In any programming environment, application developers use *external services* in addition to language intrinsics. These give access to features outside of the core language semantics.

From the language implementer's perspective, there is an additional qualitative distinction between *autonomous library services* which capture common algorithms or program patterns as sub-programs or functions *expressed only using the language and other library functions*; and *system services* which requires specific knowledge about the underlying computing system. For example in C the string comparison function `strcmp` is autonomous, whereas the `malloc` heap allocation function is a system service because it either requires to know about the address space layout of the actual hardware, or requires support from a virtual memory manager like POSIX's `sbrk` or `mmap`. This distinction is useful because the definition of autonomous services requires only a dialogue between the application developer and the language implementer, whereas the definition of system services needs a dialogue between both these parties *and* the underlying system's provider.

Which parties are involved, and how many, between the language implementers and the hardware provider constitutes a distinguishing feature of a *computing ecosystem*. For example, with the first High Performance Computing (HPC) ecosystems and the early personal micro-computer markets, there was no intermediate party between the language implementer and the hardware provider. The latter provided detailed knowledge about the hardware interface to both language implementers and application developers. This situation still exists with small embedded systems, especially those based on microcontrollers. In contrast, the IT service provider ecosystems of the early 21st century feature tens of parties involved in defining abstractions between the hardware platforms and the actual applications visible to customers.

5.2.1 Dialogue within ecosystems

The main dialogue between an ecosystem's actors considered here is the user-provider relationship, illustrated in fig. 5.1.

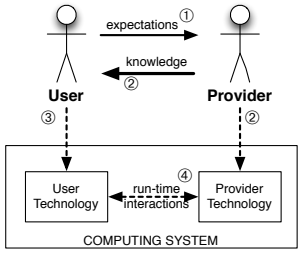


Figure 5.1: User-provider relationship in a computing ecosystem.

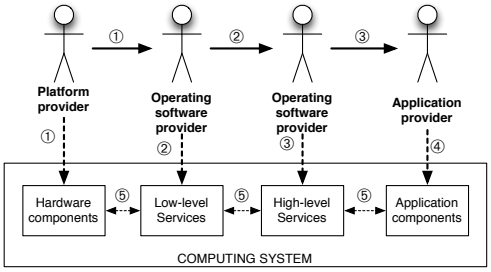
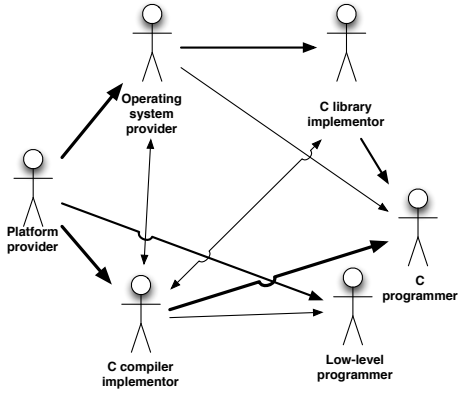
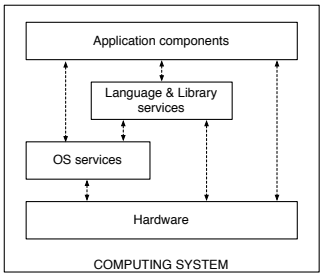


Figure 5.2: Idealized, non-realistic vision of the "abstraction stack."



(a) User-provider dialogues. The strength of the arrows represents the amount of knowledge provided.



(b) Run-time interactions between technologies.

Figure 5.3: Subset of actual ecosystem interactions in the IT service industry.

In this dialogue, a *user* party expresses wishes, or *expectations*, which are answered by a *provider* party in the design and implementation of their technology. The provider's design is then described to the user, who can subsequently use this *knowledge* to implement more technology. A *computing system* results from an actual composition of the technologies. *Run-time interactions* between components are only eventually possible if the *dialogue* between the parties was successful.

It may be tempting to describe a computing system as a linear, *stacked* relationship between components of increasing levels of abstractions (fig. 5.2). In this idealized vision, the actors in the ecosystem form a chain where the actor at one level of abstraction plays the role of user for the actor at the level immediately lower, and provider for the actor at the level immediately higher.

In reality however, ecosystems are more complex and a strict abstraction stack cannot be clearly defined. The example in fig. 5.3 demonstrates that some actor pairs are both user and provider to each other. For example, C compiler and library implementers are thus typically co-dependents. Moreover, some actors will place expectations and require knowledge from multiple providers. For example application programmers may place expectations, and require knowledge from, both the compiler and library implementers.

Despite this diversity, we can recognize in most ecosystems:

- the *platform providers* who have a privileged role as universal providers, as they do not *require* knowledge about the technology of other actors to provide their own technology (although they may advantageously exploit such knowledge when it is available);
- various *operating software providers* who do require knowledge about the platform and whose technology is not directly visible by the eventual external users of the running computing systems.

5.2.2 The need for a target audience

When pushing innovations from the hardware designer's perspective, which really is a part of the platform provider's perspective, it is thus essential to determine or choose a target ecosystem, so as to identify which parties are responsible for providing the first-level interface to the platform. Depending on the ecosystem, these *first level parties* will be either application-level programming language implementers, operating system providers, platform virtualization providers, or possibly others, all of whose will have different expectations and customs that must be acknowledged in the dialogue.

This is the obstacle that creates the HIMCYFIO pitfall introduced in chapter 1: often innovators resist choosing a target ecosystem, and thus fail to tailor their dialogue to the specific culture of their first level parties who have direct access to the innovation in the abstraction stack.

In this chapter, we propose a dialogue scenario to advertise architectural innovations, and we illustrate this scenario with the case of hardware microthreading introduced in part I.

5.3 Proposed dialogue scenario and opportunities

The *integration* of architectural innovations involves both gaining the interest of the various stakeholders and sketching an *integration path* for them. This must outline which concrete actions are required to perform the integration and fit the innovation into the ecosystem.

For this, two different perspectives must be adopted simultaneously. The first must look backwards in time and provide "adequate" support for legacy applications. This must acknowledge the perception of existing user communities, their assumptions and their culture. What constitutes "adequate" backward compatibility can be negotiated based on contemporary expectations: for example, the simple preservation of standard system Application Programming Interfaces (APIs) instead of full binary compatibility has become acceptable for commodity computing in the first decade of this century, whereas it was frowned upon ten years earlier. The second perspective must look forward and define an attractive toolbox for new applications. This must highlight issues with past customs of the ecosystem and illustrate new opportunities to gain traction. This is where a new feature can be advertised, for example hardware support for concurrency management.

This latter perspective is well-understood and usually spontaneously adopted by innovators. However, the first perspective should not be adopted as an afterthought: innovators who may resist acknowledging the full scope of their audience in the ecosystem may not recognize that it is their responsibility to sketch the integration path as an extension of current views.

□ To sketch an initial integration path, we suggest in a first iteration to mingle with the community and imitate previous integration scenarios from other successful innovators. We have three motivations for this.

One is that using a known scenario makes it easier for parties to recognize and estimate the work required to adopt an innovation. For example, porting an existing operating system to a new hardware platform is a well understood undertaking, whereas designing an entirely new operating system and then its backward compatibility layer involves high risks in management and planning.

Another is that reusing an existing scenario allows audiences to easily differentiate the innovation from its extra integration steps. For example, while porting an operating system to a new hardware platform, the new hardware features will be readily recognized by comparison with the other platforms where the operating system was running previously; whereas if, to expose a new hardware platform, a new operating system is developed, the benefits of the former may be occluded by design or engineering defects in the latter.

Our third motivation, perhaps the strongest, is that reusing a known scenario will minimize the effort to reach the first interested parties and encourage them to provide *early feedback*. This is essential as early feedback may highlight key issues with the innovation that must be addressed before effort is invested into a larger, long-term integration plan.

5.4 Case study: hardware microthreading

5.4.1 Target ecosystems

We have examined possible ecosystems for the innovation from part I, summarized in table 5.1. This table identifies for each ecosystem the parties interacting directly with the platform provider and evaluates the estimated acceptability of the proposed hardware features.

From this analysis we can recognize an interesting target: the community of IT service providers, more specifically type II and III service providers[1] who build upon Free and Open Source Software (F/OSS).

[1] As per [RH11]: type I service service providers are those providing IT services to only one client,

Ecosystem↓	First level parties in direct dialogue with the hardware provider	Preference for homogeneous, fungible chip structure rather than heterogeneous, specialized components	Tolerance of changes to the machine interface, forcing recompilation of programs
HPC	Both performance language implementers and application providers	usually	increasing
Desktop computing	Both application providers and operating software providers	yes	unusually, but increasing
Mobile computing	System integrators	unusually	yes
Video game consoles	Game designers	unusually	unusually
Embedded systems	Application providers	no	yes
Type I IT service providers	Operating software providers	yes	unusually
Type II/III IT service providers	Operating software providers	yes	usually

Table 5.1: Possible target ecosystems for the proposed architecture.

The candidate ecosystems are taken from those most likely to benefit from the availability of large amounts of parallelism on chip. This excludes e.g. domestic appliances.

Type II/III service providers are particularly interesting for two reasons. First, this community does not typically expect specialized computing platforms, i.e. its members expect balanced *general-purpose* platforms that they can specialize differently in software for different customers. This contrasts with type I providers who may consider specializing to the needs of their unique client. Type II/III providers are thus direct consumers of the "stem cells" of computing that we recognized in section 1.3; this ecosystem is therefore likely to be accepting of, or interested in, hardware designs that are not tailored to a specific application.

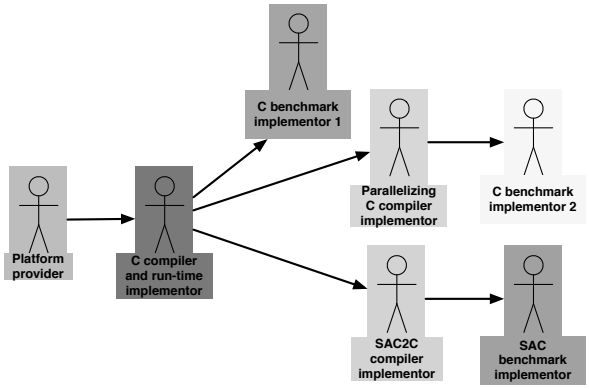
The second reason is that this ecosystem's market dynamics do not allow products to crystallize into aging legacy that must be preserved and maintained over time. This is because the lifetime of the associated business contracts does not typically exceed the lifetime of the platforms used. To the contrary, parties in this ecosystem welcome most technology developments as a positive factor for business differentiation. They are thus accepting of significant platform developments, e.g. new ISA features that require recompilation of program code, new filesystem technology that requires data migration, etc. This is useful since the proposed architecture does not propose to maintain backward binary compatibility of application code with previous platforms.

The subset of F/OSS-oriented service providers [GA04] is also particularly interesting for two reasons. The first is that the F/OSS movement is now widely acknowledged as a driving force for innovation in software ecosystems [Lev84, Web00, LT02, vHvK03, LW03, LT04, BS06, CHA06, Rie07]. Moreover, in the first decade of this century, F/OSS has also become the primary source of first level infrastructure for the online IT service industry. It must thus be acknowledged by hardware architects who want to partake in horizontal innovation networks [vH07]. The second reason is that F/OSS, by definition, promotes openness and transparency between the layers of the abstraction stack. This creates an opportunity, from the hardware architect's perspective, to receive more feedback on the promoted innovations from all parties of the ecosystem.

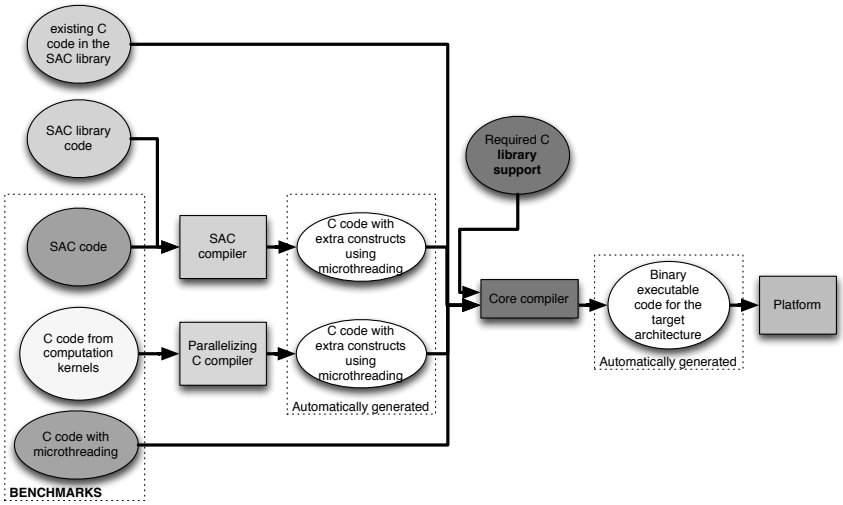
5.4.2 Dual ecosystem

While we were eventually able to recognize the IT service industry as a particularly relevant candidate ecosystem, this recognition only clearly occurred in hindsight; *during* our work,

typically within their organization; type II service providers are those providing IT services to multiple business units within their organization; type III providers cater to multiple external customers.



(a) Ecosystem actors and user-provider dialogues.



(b) Interactions between technologies.

Figure 5.4: The Apple-CORE ecosystem and its technology.

our choices were merely intuitively and subjectively motivated by past professional experience with this ecosystem. Instead, our immediate target was a project consortium named Apple-CORE, which we depict in fig. 5.4, and which shares design objectives (performance, efficiency) with the HPC community. In this ecosystem, four academic organizations play the roles of seven actors, identified as follows:

- a platform provider implementing the proposed architectural features in FPGA and emulation;
- an operating software provider for a machine interface language, extending C with primitives to control the new hardware features;
- another operating software provider for the SAC2C compiler, generating C code from Single-Assignment C (SAC) [GS06];
- another operating software provider for a parallelizing C compiler;

- three application providers for: existing benchmarks using plain C, SAC benchmarks, and hand-coded benchmarks using the proposed C language extensions.

Retrospectively, under the combined influence of both this ecosystem and the IT service industry identified earlier, our work has targeted a "middle ground" between the two. We acknowledge this as a shortcoming, because it prevented us from making choices that would have permitted performance improvements at the expense of portability, as would be appropriate in the HPC community, and choices that would have increased portability of the eventual platform to a larger diversity of application providers, as would be appropriate in the IT service industry.

Nevertheless, as we show in part III our choices were eventually successful at gaining early feedback, as per section 5.3, on the architectural innovation.

5.4.3 Ecosystem actors and their expectations

Our suggestion from section 5.4.1 was to target the ecosystem of IT service providers building upon F/OSS. In this community, the first level audience from the platform provider's perspective is constituted by:

- F/OSS *operating system providers* (e.g. Solaris, GNU/Linux, BSD);
- F/OSS *system-level language implementers* (e.g. GNU Compiler Collection, LLVM);
- more recently, the *providers of virtualization platforms*, either encapsulated (e.g. Java, .NET) or para-virtualized (e.g. Xen, KVM, VMWare).

The diversity of actors in these categories has historically caused the community to self-organize around community-based *standards* that document their provided technology, in particular:

- codified ISO and IEEE specifications for the C and C^{++} languages [II11b, II11a] and the POSIX system interfaces [IEE08, II09];
- informal consensus about commonly supported extensions to these specifications; for example, the BSD network socket API and the GNU extensions to the C language.

With standards serving as middle ground with any downstream actors in the ecosystem, innovators cannot simply make deals with particular application providers to implement their own operating system and exploit their innovation directly; they must instead either provide one of the existing standard interfaces to their proposed technology, or argue to the community why the existing standard is inadequate and should be extended.

In the latter case, standardization of new interfaces first requires *multiple* proof-of-concept system implementations of the proposed interface by *independent actors*, then a proposal for a new standard, then community evaluation of the proposal. For example, this is the process that enabled the eventual integration of thread management in [II11b] compared to the previous [II99]. This process, while lengthy, promotes peer review and avoids tight vertical integration between hardware providers and specific application vendors.

In contrast, the *lower interface* between platform providers and the first level software actors is not codified and a large diversity of platform designs exists. This is a boon to hardware innovators, because it creates a tolerance for new platforms, and it pushes existing F/OSS operating systems to design for *portability*. This in turn significantly reduces the effort required to adapt an existing code base to a new hardware design.

To summarize, the audience identified above expects the platform provider to interact with first level software actors towards providing a first implementation of the standard interfaces. It also expects the platform provider to simultaneously explain to third parties how they could exploit the platform on their own to deliver other implementations of the standard interfaces.

5.4.4 Strategy

In this context, a strategy to advertise the innovation from part I in this ecosystem would start with defining a hardware platform around the technology, then reuse existing F/OSS operating software components to provide (some of) the established standard interfaces to application developers, and then explain how this was done so that third parties could reproduce and/or extend the work.

We spell out these steps as follows:

1. describe the hardware/software machine interface;
2. define a hardware environment as will be observed by operating software components, including I/O devices;
3. implement a *freestanding* C language environment [II11b, 4§6, 5.1.2.1] which can compile autonomous code without external software dependencies;
4. port some operating system components using item 3 to provide hardware access to software;
5. port some C library components using items 3 and 4 to define a *hosted* C language environment [II11b, 4§6, 5.1.2.2];
6. document any unusual/new specific aspect of the platform required by third parties to reproduce steps 3 to 5;
7. illustrate with example/evaluation programs that exploit items 3 to 5.

The narrative of our dissertation essentially follows this plan. Part I, in particular chapter 4 covered item 1. We then address item 2 in section 5.5 below. Chapter 6 covers items 3 to 5. Chapters 7 to 11 address item 6. Item 7 is finally covered in part III.

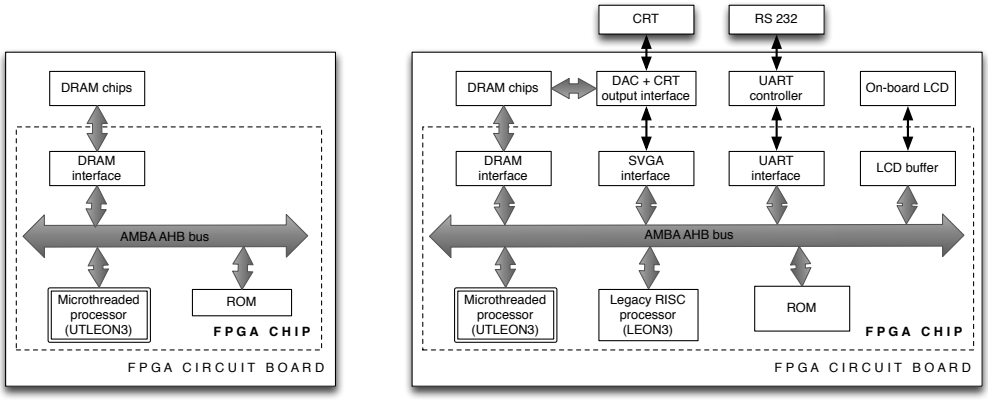
5.5 Platform definition

To support the implementation steps sketched above, a platform with memories and I/O devices is needed around the processor chip.

The initial hardware environment implementing the architectural concepts from part I, available prior to our work, was extended as depicted in figs. 5.5 and 5.6. This equips the prototype microthreaded chip with an I/O system that can be connected to the necessary hardware devices, or low-level emulations thereof. More specifically, the FPGA integration was performed by another research group [DKK+12], and we extended the emulation environment to mimic the FPGA integration.

5.5.1 Support for I/O and legacy system services

Early on we observed that existing operating system code could not be reused directly with the proposed microthreaded architecture, even equipped with I/O devices. Indeed, all existing reusable operating system codes we could find, from embedded to server platforms

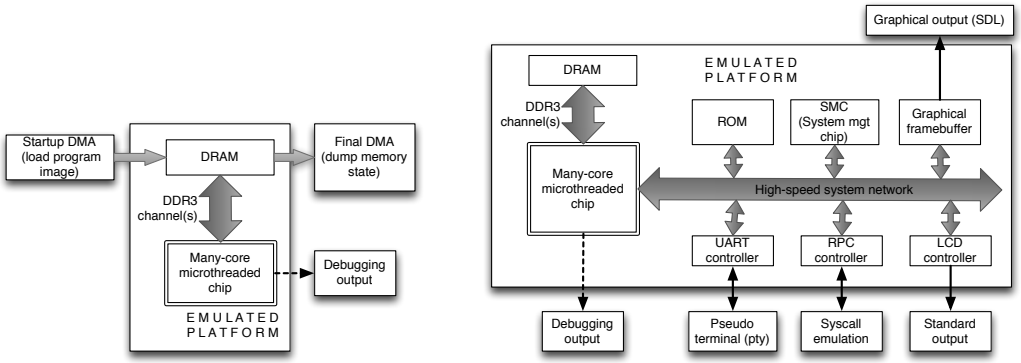


(a) Initial FPGA platform.

(b) FPGA platform suitable for software integration.

Figure 5.5: Extension of an FPGA prototype into a complete system.

This extension was performed by other researchers [DKK+12].



(a) Initial emulation platform

(b) Extended emulation platform suitable for software integration.

Figure 5.6: Extension of an emulation platform into a complete system.

This extension was performed as part of our work.

and from monolithic kernels to distributed systems, require support for *external control flow preemption* (traps and interrupts) at the pipeline level for scheduling, and either *IPIs* or *on-chip programmable packet networks* for inter-processor control and system-level communication. Meanwhile, the proposed design from chapters 3 and 4 omits preemption from the core pipeline to favor HMT instead. It also omits IPIs to favor a custom NoC supporting active messages [vECGS92] instead.

Because of this mismatch, any effort to port an existing operating system would require the system designer to either introduce the missing features into the architecture design, with the risk that these would impact performance negatively and add complexity to the machine interface, or redesign and re-implement the operating system components, which would add another significant development expenditure to the porting effort.

□ Instead, we took a transverse approach which we detail in [PLU+12]: add one or more *companion* processors implementing a legacy architecture to the platform, and use them to run one or more instances of an existing operating system. We call this *heterogeneous integration*. Once companion cores are available, it becomes possible to *delegate* any uses of system services from application code running on the new architecture to the companion processor(s).

The integration of a companion processor can be achieved either on-chip, i.e. on the same die, or off-chip, i.e. in a different chip package, in either of two ways:

- the companion processor shares the same memory system. In this case, the arguments and results of system services can be communicated implicitly through the cache coherency protocol. This is the approach taken in fig. 5.5b;
- the companion processor is a device on the I/O interconnect. In this case, the arguments and results of system services must be marshalled. This is the approach taken in fig. 5.6b.

The integration with shared memory provides the most flexibility for software and a higher bandwidth, but requires that the legacy core design can integrate with the memory network. Whether this is possible depends on the selected design: the address width, endianness and layout of data structures by the compiler must be compatible. In contrast, the integration on the I/O interconnect provides the most flexibility for the system integrator at the expense of increased access latencies and reduced bandwidth due to the need to marshal arguments and results.

Nevertheless, in either case, the porting effort for system services becomes minimal. The only new implementation required is a set of wrapper APIs on the microthreaded processors that serve as a *proxy* for a syscall interface implemented on the companion processor.

Although we developed this approach independently, we later recognized it in the "multikernel" approach proposed by the designers of Barrelfish [BBD+09], and at the larger scale in the use of AMD Opteron chips to serve as "service processors" in Cray's XMT next to the main grid of simpler MTA cores [Kon07].

5.5.2 System architecture overview

From the software perspective, the platform is structured as follows:

- the microthreaded chip is connected to external memory and I/O devices through a system interconnect;

- memory and I/O devices either share the same interconnect (e.g. fig. 5.5b), or memory uses a separate network (e.g. fig. 5.6b, cf. also section 3.4.1);
- a configurable Read-Only Memory (ROM) contains initialization routines and data structures that describe platform parameters; for example, the number of processors, the system topology, the access protocol on the NoC to the companion core(s), the location of devices in the address space, the cache sizes, the RAM size;
- there is at least one I/O device suitable to define an interactive console (e.g. a Universal Asynchronous Receiver-Transmitter (UART));
- there is at least one I/O device able to measure real time and trigger asynchronous events at specified times (e.g. a Real-Time Clock (RTC));
- there is at least one I/O device suitable to exchange larger data sets with the outside world (e.g. disk, network interface, etc.);
- there are one or more companion processor(s) running a legacy operating system for system services that cannot be implemented on the microthreaded cores;
- upon start-up, the microthreaded processor automatically starts a thread running the initialization code from ROM.

In the FPGA environment, the ROM contents are defined while configuring the fabric; in the emulation environment, the ROM contents are initialized from file in the host environment. Other I/O devices, like a graphical frame buffer and matrix displays were also implemented but will not be considered further here.

Summary

Avoiding the HIMCYFIO pitfall, and marketing architectural innovations into existing computing ecosystems require acknowledging both the current culture and assumptions of the candidate audience. The *integration* of new features into usable systems should avoid investing in entirely new operating software code bases and instead benefit from reusing the existing knowledge and software code base of the ecosystems.

□ This in turn may constrain the hardware designer to provide some backward compatibility with past concepts. While this may feel like a restriction, it is also an opportunity to gain an early audience. Most importantly, any *fundamental* design issues will be revealed by legacy-oriented audiences just as well as they would be by custom, future-oriented new software stacks; meanwhile, they can be revealed *earlier* by targeting first existing ecosystems and taking their viewpoint.

□ Applying these guidelines to the architecture from chapters 3 and 4, we have proposed in section 5.4 two candidate ecosystems and a strategy to achieve integration in this context. This in turn requires the definition of a hardware platform, which we provided in section 5.5, and allows us to plan software integration steps which will be followed by the following chapters.

Chapter 6

First level programming environment
—A "quick and dirty operating system" for microthreaded chips

> If it's a good idea, go ahead and do it. It is much easier to apologize than it is to get permission.
>
> Admiral Grace Hopper

Abstract

In this chapter, we describe how to provide an implementation of C to the new proposed architecture. We use non-standard features from GNU CC in an effective, yet simple way to generate code for the new architecture. We also define a simple input syntax to control the concurrency management primitives of the target machine, which we call "SL." We then generalize our approach as a compiler combinator which can extend other code generators to recognize our input syntax. Finally, we use our technology to port C library and operating system services to the platform introduced in chapter 5.

Contents

6.1 Prior work

○ Prior to the work described in this dissertation, some research had been realized to design a C-based interface to the proposed architecture. This work was advertised starting with [Jes06a] and culminating with [Ber10]. At the start of our research, we carried out an extensive analysis of the design and semantics of this proposed interface, which we detail in Appendix G.

□ To summarize, this language was based on a few language primitives that semi-transparently expose the ISA semantics introduced in chapter 4. However, through analysis we were able to identify the following shortcomings, not described in previous work:

- the proposed design only implements a subset of C, which conflicts with the requirements described below in section 6.2.1;
- it introduces significant complexity in the front-end and middle-end of a C compiler, by requiring obtuse abstract semantics for the handling of synchronization channel endpoints. We describe this in Appendix G.2;
- it *cannot* be compiled to first generation interfaces (cf. section 4.7), which prevents its applicability to the UTLEON3 platform which we intended to support. We prove this in Appendix G.3.

While we attempted initially to fix the design with minor semantic restrictions to extend its applicability to all our target platforms, we were not able to find a satisfying simple solution to all these shortcomings. We thus chose early to not invest further effort in this direction. Instead, we point out that these shortcomings are a consequence of a research strategy that attempts to design a language prior to and independently from its prospective implementations in a compiler.

6.2 Guiding considerations for an interface language

6.2.1 Choice of C, language subset or superset

The choice of C as a starting point, as opposed to some other language, can be debated. Indeed, C presents the burden that it was designed for mostly sequential computers, and the programmability and generality of the proposed architecture design could be demonstrated using languages designed with concurrency in mind, for example Erlang [AVWW96] or Google's Go [Goo].

Yet it is customary to implement C on new architectures. This is motivated first by the ubiquity of C implementations that can be customized, the wide audience for the language, and its *de facto* status as the low-level hardware interface language for the audiences identified in chapter 5 (including its role in implementing portable operating systems). Beyond these practical motivations, we found that there are four fundamental properties of C that are relevant in architecture research:

1. C is not a fixed language *defined* by specification; it is a family of languages, each defined by the *actual capabilities of its implementation*, and the *intention for interoperability between implementations*.

Side note 6.1: Contenders to C to pitch new architectural designs.

LLVM IR/bytecode. This is a strong contender with C, however its developer communities still lack the multiculturalism and distributedness of existing C communities as of this writing;

CLI (.NET) / JVM. These are not multifaceted, not lean, not multicultural, not distributed, and place strong requirements on the underlying platform: mandatory isolation, mandatory preemption, mandatory coherent caches, etc.;

C++. Its language design derives from C, and most C++ compilers contain parts of a C compiler as a substrate. The strategy we describe in the rest of this chapter rely on the parts of a C compiler common with C++ compilers, and could thus be applied to C++ with no changes.

2. Those shared semantics of most C implementations that are used as reference by programmers, and thus collectively named "standard" and captured in [II99, II11b], *carefully and deliberately avoid making assumptions about features available in the hardware machine* underlying any specific C implementation (any exceptions, e.g. required support for preemption in `signal`, are encapsulated in library services, i.e. outside of the core language).
3. Moreover, the C language is purposefully designed for *hardware transparency*, that is, the language semantics do not attempt to hide the hardware machine interface.
4. All implementers of C are allowed to *extend their implementation with new features* not found in other implementations; these can then be publicly scrutinized and discussed by international communities of industrial, academic and private experts. Successful extensions are usually adopted by other implementers and become candidate for future standardization.

It is this combination of well-structured, multifaceted and lean semantics, an implementation-driven community, a tolerance for and promotion of hardware diversity, and a multicultural, distributed meritocratic approach to new developments, not found combined in any other language, that makes C unique and especially desirable for hardware architecture research.

We were asked to comment on the opportunity to support only a restricted subset of the C language in a new implementation. This seems interesting because any specific set of evaluation or benchmark programs only needs the features sufficient to compile them, and restricted language support may reduce the amount of testing required before releasing tools for public use. In response, we advise strongly against exploiting this opportunity. While a language subset may be sufficient to compile small test programs, ultimately the run-time environment also requires library and system services whose code typically exploits most features of C. In particular, if an implementer wishes to reuse existing library or operating system code, an extensive implementation of C is needed with support for common non-standard extensions. This is the perspective we took in our work.

6.2.2 Language primitives vs. APIs

Once new features are introduced in a machine interface, two ways exist to exploit them from programs: *encapsulation* in APIs and *embedding* into the language via new primitive constructs recognized by compilers and associated new translation rules to machine code[1].

[1] In the case of C, extensions via the preprocessor's `#pragma` feature is a form of embedding as it influences code generation.

Encapsulation is technically trivial, and it is desirable when porting existing software using established APIs such as the POSIX thread interface. However the following must be considered with the proposed architecture. As explained in chapter 4, a thread that issues a long-latency asynchronous operation, e.g. memory load or thread creation, uses regular ISA register names for the endpoints of the communication channels with the asynchronous operation. Meanwhile, the proposed interface creates the opportunity to interleave the asynchronous thread management operations (e.g. "allocate," "create") or inter-thread communication operations with other instructions *from the same thread*. To exploit this opportunity with encapsulation, the two phases must be part of separate API functions, and code generators must be configured to avoid reusing these ISA register names for computations while a thread management or communication operation is ongoing. Otherwise, any register spills between phases will cause the thread to wait prematurely for completion of the operation and waste an opportunity for overlapping instruction execution with the asynchronous operation. The same applies for synchronizers that hold the future of asynchronous completions (section 4.2): if the synchronizer that holds a future is spilled, this would cause the creating thread to wait prematurely on the asynchronous operation. To address these issues, a code generator would need to perform an inter-procedural register allocation; furthermore, if the API implementation is compiled separately from the application code, register allocation must then be deferred until all objects are available. Given that no publicly available compiler framework had support for link-time inter-procedural register allocation prior to our work, encapsulation seemed impractical with the new architecture and embedding remained as the unavoidable strategy.

That said, there is also a quantitative reason as to why embedding is more desirable. The architecture allows fine-grained, short-latency thread management and inter-thread communication; and the cost of diverting the control flow for a procedure call is large compared to the synchronization latency (e.g. 40 processor cycles to transfer control to a different procedure vs. 6 cycles to create a family of one thread and 0-1 cycle to communicate a scalar value from one thread to another). In this circumstance, the choice to embed the TMU primitives as new language primitives reduces the overhead to exploit the synchronization and scheduling granularity offered by the architecture.

6.2.3 General design directions

One design motivation that supported our work was to entice code generators and programmers to *expose the fine-grained concurrency* of numerical computations, even the partial concurrency available in dependent computations, in order to enable the automated mapping in hardware of all program fragments, even a few instruction long, to separate cores or hardware threads. This requires, for example, simplifying the replacement of *sequential loops* in programs by a construct that a) isolates simpler, *non-repetitive communicating sub-sequences*, b) instructs the underlying hardware to run the fragments on separate, *simple* processors/threads and c) instructs the underlying hardware platform to *connect* the processing units in a network of dataflow channels that corresponds to the computation structure. When this design objective is reached, it becomes possible to transform dependent loops in sequential programs by a network of dependent threads interleaved in the pipeline, removing the need for branch prediction to maximize utilization.

Another motivation was to *promote resource-agnosticism*, that is promote the expression of programs in a style where the semantics stay unchanged should the hardware parameters evolve. In particular, the approach should discourage programmers from assuming,

or knowing, or restricting at run-time the specific amount of effective parallelism (e.g. the number of processors or thread contexts available) when constructing algorithms. This is because otherwise the program is tailored to a specific hardware topology and must be re-designed upon future increases of parallelism. This requires language mechanisms that can express concurrency mostly via *data dependencies* and *declarative concurrency*, in disfavor of explicit control of individual thread creation and placement, and explicit inter-thread communication. When these features are used, it becomes possible to scale the run-time performance of a program by changing the amount of parallelism, and without changing the machine-level encoding of the program. Conversely, it also becomes possible to run any concurrent program on a single processor, since the program cannot assume a minimal amount of parallelism. Thus the flexibility required for dynamically heterogeneous systems (cf. chapter 2) is achieved.

We detail this second objective further in [PJ10]; they are shared with other language designs, such as Cilk [BJK+95] or more recently Chapel [CCZ07].

6.2.4 Mandating a sequential schedule

We considered the hypothetical situation where there were no theoretical or research challenges to the implementation of code generators and operating software services. We then considered what were instead the likely practical obstacles to evaluation of the platform. The motivation was to determine early on what the likely obstacles were, then integrate their avoidance as technical requirements.

We found that the *validation* of the infrastructure, which is the step immediately prior to evaluation, was the largest obstacle. Validation ensures that the various tools function as expected; that is, for each tool we can check for:

completeness: valid input is accepted successfully;
correctness: valid output is produced consistently from valid inputs;
consistent rejects: invalid inputs are rejected consistently with informative messages.

By negating any of these aspects, we can enumerate all the possible *failure modes* which could be observed in a software stack with higher-level compilers targeting a C interface, in turn targeting the hardware platform:

incompleteness errors :

- ‹F1› the higher-level compilers fail clearly on valid benchmark program;
- ‹F2› the code generator fails clearly on valid input source code;
- ‹F3› the assembler / linker fails clearly on valid input assembly;
- ‹F4› the reference hardware implementation fails clearly on valid machine code;

incorrectness errors :

- ‹F5› the higher-level compilers accept valid benchmark programs and produce invalid code in the interface language;
- ‹F6› the code generator accepts valid input source code and produces invalid assembly code silently;
- ‹F7› the assembler / linker accepts valid input assembly and produces invalid machine code silently;
- ‹F8› the reference hardware implementation accepts a valid machine code and produces incorrect outcomes;

inconsistency errors ("false positives"):

‹F9› the higher-level compilers do not reject invalid benchmark programs and produce apparently valid code with invalid behavior in the interface language;

‹F10› the code generator does not reject invalid input source code and produces apparently valid assembly with invalid behavior;

‹F11› the assembler / linker do not reject invalid input assembly and produces apparently valid machine code with invalid behavior;

‹F12› the reference hardware implementation does not reject invalid machine code and produces unclear outcomes;

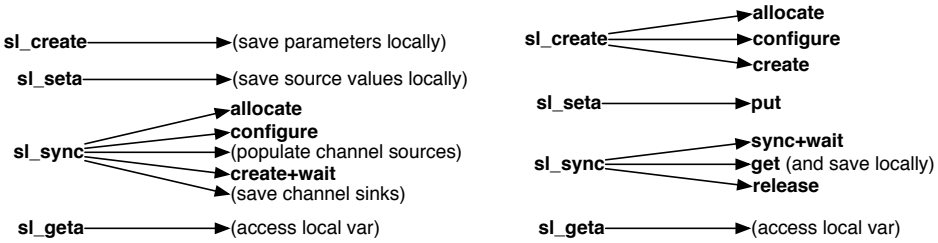
compound failures : any combination of the above.

Based on personal prior experience with complex infrastructures, we established the need for clear methods to isolate failure modes and the components of compound failures as a primary requirement for any technical realization. In particular, we put the following goals at the highest priority:

- we needed to be able to clearly separate a) compounds of incorrectness errors upstream and consistent rejects downstream from b) incompleteness errors. This is crucial to establish responsibility upon failure. For example, if an error is observed at the code generator, the cause of the error can be either failure mode ‹F5› or failure mode ‹F2›. In the former case, the responsibility lies with the provider of higher-level compilers, in the latter case with the interface language or below.
- we needed to be able to clearly separate a) incorrectness errors at one level from b) compounds of incorrectness errors upstream with a stack of inconsistency errors downstream. This is crucial to estimate the cost to fix the error. For example, if an error is observed at the hardware implementation, the cause can be either failure mode ‹F8› or a combination of failure modes ‹F5› and ‹F10› to ‹F12›. The pitfall is that peer operating software providers may be tempted to believe failure mode ‹F8› and push the responsibility for the error entirely towards the platform, whereas work at all levels is actually necessary to resolve the situation.

Retrospectively, these concerns were not completely new. At the level of the hardware implementation, both the partners in charge of UTLEON3 and the author of MGSim had set up an environment where any machine code could be repeatedly run over multiple points in the hardware design space (e.g. varying number of cores, various memory architectures). This could distinguish invalid machine codes, i.e. "software errors" which would fail consistently across all design points, from incorrectness or incompleteness errors in the hardware implementation, i.e. "hardware errors" which would fail inconsistently across hardware design points. More generally, to support more detailed failure mode detection upstream across software layers, extensive unit test suites and regression tests are needed.

Beyond unit testing, we decided to place an extra requirement on any C language extensions: ensure that the C compiler can generate valid *sequential* code for any new language constructs towards existing (legacy) downstream tools and architectures, e.g. commodity desktop computers. This enables proper troubleshooting and analysis of behavior on existing computers. This requirement is crucial to validate the correctness of the interface-level test suite itself, and also to robustly distinguish inconsistency or incorrectness errors upstream, under the responsibility of the higher level compiler providers, from any incorrectness or incompleteness errors downstream, under the responsibility of the platform. This is akin to requiring Cilk's *faithfulness* [Lei09] or Chapel's *serializability* [Cra11].



(a) Translation with fused creation.

(b) Translation with detached creation.

Figure 6.1: Translation of SL primitives for bulk creation (simplified).

We describe the difference between fused and detached creation in section 4.3.1.2.

6.3 Proposed code generator

After an initial step where we crafted various unit test suites as per the requirement above, we attempted to realize the semantics of the desired language constructs using an unmodified code generator and inline assembly: defining a thread program, defining dataflow channel endpoints, using family creation instructions from the ISA.

□ Using a serendipitously powerful combination of existing language features from the GNU C Compiler (GNU CC), we successfully implemented a freestanding C translation environment [II11b, 4§6], with extra language constructs to access the new hardware concurrency management features. We detail this experimental process and the corresponding code generation strategy in Appendix H.

To summarize, our implementation is based on a front-end with three phases. The first phase occurs between C pre-processing and translation: *new syntax forms* that look like function calls, i.e. keyword followed by parameters between parentheses, are substituted using context-free rules in M4 [KR77]. The substituted C code uses GNU CC extensions, including inline assembly. Then GNU CC's legacy code generator towards the substrate ISA, i.e. without extensions, is invoked with flags to instrument the register allocation and control the visible synchronizer windows (cf. sections 3.3.3 and 4.3.3). Finally, a post-processor modifies the assembly code generated by GNU CC to adhere to the ISA semantics, and extends it with control bit annotations for thread switching (section 4.4).

The new syntax forms thus form new *language primitives* that extend the C substrate language. These language primitives expose fewer semantics than allowed to programs by the machine interface (chapter 4); they also intentionally erase the distinctions between the various hardware implementation variants, so as to be translatable to all of them from a single source representation.

□ The resulting compiler technology is able to input a program source expressed once using this interface language, and emit code for various implementations of the new architecture and for a legacy sequential processor, as required in section 6.2.4. We call this interface language "SL," which we describe in detail Appendix I, and whose main constructs and their translation are illustrated in table 6.1 and fig. 6.1. We further ensured that the main command-line tool that controls the transformation pipeline, called `slc`, has the same interface as an existing compiler command (`gcc`); this way, we were later able to successfully reuse it as a drop-in replacement in existing build systems for legacy software, for example existing C library code needed to run benchmarks as described below in section 6.4.

Construct	Description

`sl_def(`*name*`,,` *channels...*`) {`
body...
`} sl_enddef`
	Define a thread program. The signature declares a static set of channel endpoints.

`sl_create(`*range*`,` *name*`,`
channels...`);`
`...` `sl_sync()`
	Perform bulk creation and synchronization on a named thread program with "source" values for the dataflow channels. Both parts form a single syntactic construct.

`sl_glparm(`*type*`,` *name*`),` `sl_glarg(`*type*`,` *name* `[,` *v*`] )`	Declare a "global" dataflow channel endpoint. Types are manifest. The "a" version optionally provisions a source value.
`sl_shparm(`*type*`,` *name*`),` `sl_sharg(`*type*`,` *name* `[,` *v*`] )`	Declare a pair of "shared" dataflow channel endpoints. Types are manifest. The "a" variant optionally provisions a source value for the "leftmost" channel.
`sl_getp(`*name*`),` `sl_geta(`*name*`)`	Read from the named endpoint of a dataflow channel.
`sl_setp(`*name*`,` *v*`),` `sl_seta(`*name*`,` *v*`)`	Write to the named endpoint of a dataflow channel.

Table 6.1: Main constructs of the resulting SL language.

A specification is given in Appendix I; example uses are given in Appendices J and K;

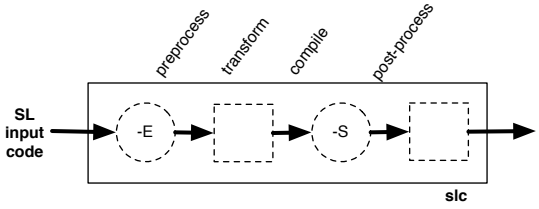
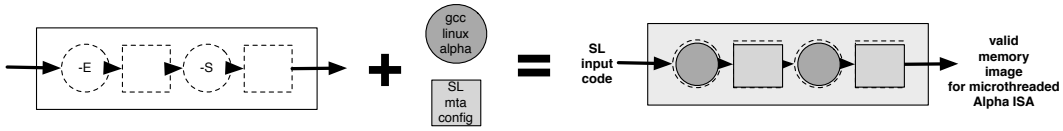


Figure 6.2: Overview of the placeholders in the SL tool driver.

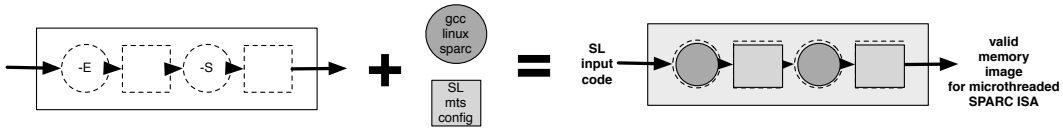
6.3.1 Compiler combinators

A remarkable aspect of the SL tool chain, one originally motivated by short-term practical requirements, is that it does not modify the underlying C compiler used to generate the assembly code from the input source. Instead of going the "traditional route" of language extensions by modifying an existing compiler and introducing new features throughout the front-end, middle-end and back-end, we instead *subverted* the compilation pipeline of the existing C compiler to "trick" it into creating valid code for the new architecture.

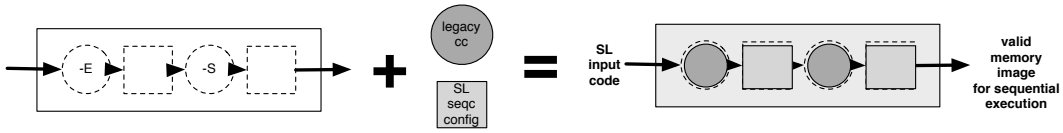
A bird's eye overview is given in fig. 6.2: we first let the underlying compiler pre-process the code, then we perform source-to-source transformations on the pre-processed code to insert the new machine constructs via inline assembly and other techniques, then we let the underlying compiler process the code, and finally we post-process the generated assembly source to ensure its validity to the target machine interface. We detail this scheme in Appendix H. We then made it simple to configure *which* underlying compiler is used and the *set of rules* for the upstream code transformation and downstream assembly post-processing.



(a) with `alpha-linux-gnu-gcc`.



(b) with `sparc-linux-gnu-gcc`.



(c) with a legacy C compiler.

Figure 6.3: Combinations of the SL tool chain with various underlying C compilers.

□ We would like to suggest that the resulting framework is a *compiler combinator*: given an existing C compiler, we can combine the existing compiler with a set of rules and our tool chain to obtain a new compiler for a different target and/or an extended input language. With this methodology, we were later able to quickly support three different implementations of the new architecture (a 64-bit platform with the Alpha ISA and two 32-bit platforms using the SPARC ISA) and sequential execution by simply changing the operands of the combinator, as illustrated in fig. 6.3.

We consider the generality of this approach, together with our ability to reuse existing code generators without changes, to be a confirmation that the new architecture's design can be considered as an add-on improvement on existing ISAs, instead of a radical new approach to programming parallel processors at a low level. As we show in the remainder of our dissertation, the originality of the proposed design is found at a higher level, with memory semantics and resource management. By choosing an implementation strategy that keeps away from complex approaches to code generation, we were able to invest more effort into addressing system-level issues. Besides, our strategy *does not sacrifice performance*: the benchmark results in chapter 13 show that our non-intrusive, blunt approach to code generation was sufficient to exploit the architecture's fine-grained concurrency management features.

The generality of the framework was confirmed again later with the successful re-targeting of the SL tool chain to other software-based concurrency management frameworks [vTJLP09, Mat10, UvTJ11] for existing hardware architectures.

6.3.2 Discussion on the generality of the approach

As we detail in Appendix H, to achieve our implementation we used standard features from [II99, II11b] and a tool box formed of carefully selected pre-existing language extensions from the GNU C Compiler (GNU CC):

- type inference with the `typeof` keyword [Fred], to produce declarations to new variables with the same type as an existing variable, for example:
  `__typeof__(a) b;`
- inline assembly with the `asm` keyword with access to enclosing local C variable from the assembly code [S.03, Frea], for example:
  `__asm__("mov %1,%0" : "=r"(b) : "r"(a));`
- explicit machine registers for local variable declarations [Free], for example:
  `int a __asm__("$1");`
- the ability to exclude selected register names from register allocation [Frec], for example using the command-line argument `-ffixed-$1`.

We investigated whether our implementation was tied to GNU CC due to the use of these non-standard language extensions, or whether the strategy could be adapted to other compiler technologies. The purpose of this investigation was twofold. First, we acknowledge a recent trend (post-2010) that promotes LLVM as a substitute to GNU CC for general-purpose programming, and we wish to ensure a future transition path, should one become necessary. Second, we wish to highlight that the successful exploitation of the architectural features was not dependent on this specific compiler technology, so as to enable third parties to develop their own compiler technology instead.

We found active support for `typeof` also in IBM's and Intel's compilers [IBMb, Intb], and in ARM's compiler until version 3.0. Although this construct for type reuse in C variants is still relatively uncommon, we expect support for this feature to become prevalent once the new C^{++} language standard [II11a] is adopted by vendors: most industry-grade frameworks share features between their C and C^{++} compilers and the new standard mandates support for the C^{++} `decltype` keyword with identical semantics as `typeof`.

Support for the `asm` keyword has been historically pervasive across implementations of C. It is the main means provided to C programmers by the compiler implementer to bypass the compiler and emit machine instructions explicitly. We found active support for both `asm` and the use of C variables in inline assembly in particular in Microsoft's Visual C/C^{++} compiler [Cor], Oracle's (previously Sun's) Solaris Studio compiler [Ora11], IBM's XL compiler [IBMa], Intel's C/C^{++} compiler [Intc], ARM's compiler [ARMa], LLVM and its Clang front-end [LLV], and Open Watcom [Con08].

Support for explicit mapping of variables to registers is less prevalent but not unique to the GNU implementation either. We found that it is supported by Intel's and ARM's compilers using the same syntax as GNU [Intb, ARMb] and in Open Watcom via pragmas.

Finally, we also found support for customizable register sets during register allocation in Intel's compiler (`-mfixed-range`, [Inta]). The set of available machine registers is also expressed explicitly in the source code of Open Watcom's and LLVM's back-end code generators, so one could manually modify this set and generate multiple compilers with different register uses in each. We expect that this technical modification could be realized in proprietary compilers as well, if need arises.

To summarize, although we were fortunate to find this conjunction of features and their flexibility in the GNU implementation, we can reasonably expect that other compilers support similar features, or can be modified to support them at a low cost.

6.3.3 The need for manifest typing

The reason why we separate the syntax to declare integers and pointer thread arguments (`sl_glparm`, `sl_glarg`) from the syntax for floating-point (`sl_glfparm`, `sl_glfarg`) is that

```
1 sl_create(...);
2 if (cond)
3     sl_sync(...);
```

(a) Halves not in the same level.

```
1 if (cond)
2   sl_create(...);
3   sl_sync(...);
```

(b) Halves not enclosed in a block.

```
1 if (cond) {
2   sl_create(...);
3   sl_sync(...); }
```

(c) Valid use: halves enclosed in a block.

Figure 6.4: Example uses of `sl_create`...`sl_sync`.

they must be translated to different register classes in the inline assembly, and our translation strategy cannot analyze C types.

This syntax is undesirable because it exposes a feature of the underlying ISA (namely, different register classes on Alpha) that C was originally intended, by design, to hide. Moreover, this proposal is incomplete as support for C's `long double` and `_Complex` types require other register classes or register pairings on most ISAs which would require yet additional SL syntax with the current implementation.

Instead, to alleviate this issue and properly hide register classes *without introducing type analysis in the source-to-source transformer*, we suggest that future work use the new `_Generic` construct from [II11b, 6.5.1.1] which can select different expressions based on the actual type of a conditional expression. This mechanism would enable delegating the task of choosing the right implementation based on the type information to the underlying code generator.

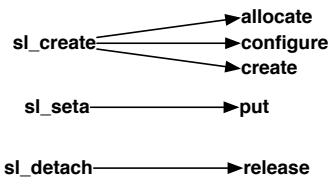
6.3.4 Why "create" and "sync" are bound

A characteristic SL feature, detailed in Appendix I.5.8.1, is the binding between the words "`sl_create`" and "`sl_sync`" in one syntax rule for C's block item. This implies, for example, that the forms in figs. 6.4a and 6.4b are invalid, whereas the form in fig. 6.4c is valid.

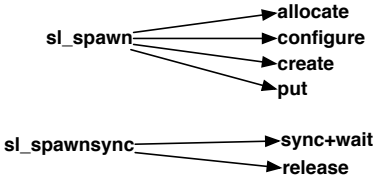
There are three motivations for this, one fundamental and two practical.

The fundamental motivation is to hide the semantic distinction between "fused creation" ISAs and "detached creation" ISAs introduced in section 4.3.1.2. To hide this, we specify that the created work starts "no earlier than `sl_create`" and "no later than `sl_sync`" in the control flow. This is to provide the understanding to the programmer that either part of the construct is ultimately responsible for the actual creation, without specifying which. If the language had allowed to omit `sl_sync`, as in fig. 6.4a, we would not be able to generate code for fused creation ISAs where the information needed for creation, generated by uses of `sl_seta`, is only fully known at the point `sl_sync` is reached. We detail this further in Appendix G.3.

The first practical reason is that each use of `sl_create` expands to both *multiple statements* at the point where it is used, and *new variable definitions* at the beginning of the current scope. The scope is identified by the last preceding occurrence of the left brace "`{`" in the program source. The new variables are used both by the expansion of `sl_create` and subsequently by the expansion of `sl_sync`. If the language would allow `sl_sync` to occur in a higher-level scope than `sl_create`, then the variable declarations would need to be pulled upwards to the first common scope. This is not possible in our setting, because some of the data types involved may be defined with `typedef` in the inner scope. A proper treatment of this would require full type analysis of the program source, i.e. a full-fledged C front-end, which is what we tried to avoid in the first place.



(a) Detached creation with lexical binding.



(b) Separated creation and synchonization.

Figure 6.5: Translation of SL primitives for detached creation (simplified).

The other practical reason is that if the language would allow the pair "`sl_create-sl_sync`" to occur in a position in the C syntax which only accepts statements (as in fig. 6.4b), we would confuse text editors and other source analysis tools which assume that the semicolon is a statement separator.

6.3.5 Opportunities for platforms with detached creation

Setting aside the aforementioned requirement to support fused creation ISAs, and focusing instead exclusively on ISAs featuring detached creation, an opportunity exists to expose this flexibility in the language interface.

We explored this opportunity as follows. First we extended the pair "`sl_create-sl_sync`" with a new form "`sl_create-sl_detach`" with the same lexical structure as described above in section 6.3.4, but where the bulk synchronizer is released upon `sl_detach` without synchronization (fig. 6.5a).

We also defined a new statement `sl_spawn` which performs bulk creation, sends source values for the dataflow channels, and produces the identifier for the remote bulk synchronizer into a variable. We then defined a new statement `sl_spawnsync` that combines both synchronization on the named bulk synchronizer and releasing the bulk synchronizer. We summarize this in fig. 6.5b. As is, this construct does not allow spawned workloads to use the "shared" dataflow channels from section 4.3.3.3, because the argument list of `sl_spawn`, necessary to declare the temporary variables to hold the channel sink values, may not be visible from `sl_spawnsync`.

Through discussions with our users, we obtained the following feedback:

- the behavior of `sl_spawnsync` is only properly defined if it is used only once by a program on a given bulk synchronizer. This is because after a first occurrence releases the synchronizer, any further occurrence would use a stale, invalid synchronizer identifier. The hardware protocol is not designed to handle this situation. Because of this, the definition of proper functional *futures* [Hal85] using `sl_spawn`...`sl_spawnsync` with multiple uses of the second half would require an additional flag to test the validity of the bulk synchronizer, which must be updated atomically throughout the system at the first run-time use of `sl_spawnsync`.
- a problem exists if the argument list of "`sl_create`...`sl_detach`" or "`sl_spawn`" does not fit entirely through hardware synchronizers. Indeed, our implementation escapes arguments to the call stack, as we will describe in chapter 8 for `sl_create`...`sl_sync`; this approach can only be used if the lifetime of the activation record of the creating

thread extends beyond the lifetime of the created thread(s). This is only true if `sl_spawnsync` is executed before the escaped arguments are cleaned up, or the created workload terminates before the creating thread. To solve this the language feature must be constrained in either of the following ways:

- `sl_spawnsync` must appear within the same C scope at `sl_spawn`; or
- `sl_spawn` and `sl_create...sl_detach` are restricted to only accept N arguments, where N is an implementation-specific constant that describes the number of channels physically available; or
- `sl_spawn` and `sl_create...sl_detach` allocate extra arguments on a shared heap. Then either the first use of `sl_spawnsync` (for `sl_spawn`), or the termination of the created work, whichever happens last, becomes responsible for the de-allocation of the argument data from the shared heap.

▷ The spectrum of solutions to these issues involve complex trade-offs between usability and performance which we did not explore fully. Lacking understanding of these trade-offs, we chose instead to not advertise these constructs as an official language feature. Instead, we used them as a system-oriented feature (i.e. hidden from applications) to implement some of the operating software services described below in section 6.4.

6.3.6 Discussion on the language design and future work

We do not claim any innovation in terms of programming language design. We acknowledge that the proposed language constructs are crude and tailored to the specific architecture considered. Their semantics are a mash-up of concepts borrowed from traditional fork-join parallelism, bulk-synchronous parallelism, Cilk, and previous work from our research group, selectively chosen to facilitate the demonstration of the proposed machine interface.

A scientific contribution should be found instead in the careful choice of syntax which enables straightforward code generation from the same source semantics to a diversity of platforms, using a relatively simple wrapper around existing compilers (section 6.3.1).

▷ Since the advertised innovation is to be found in the machine interface, it follows that a diversity of existing programming languages could be extended or ported towards the proposed architecture. For example:

- the parallel loop constructs with static scheduling from OpenMP [Ope08], independent loops in Fortran, and the parallel loops from Intel's TBB [Rei07], map transparently to bulk creation and synchronization in our platform. The variants of these constructs that require dynamic scheduling can be translated to a program using the run-time scheduler from [SM11].
- the fork-join constructs from Cilk can be mapped onto `sl_spawn`/`sl_spawnsync` such as described in section 6.3.5. The issues we identified with `sl_spawn`/`-sl_spawnsync` would not apply because Cilk forces its "sync" construct to be in the scope of its "spawn" constructs and does not allow multiple "sync" occurrences for each "spawn."
- the vector operations from OpenCL [Khr09] can be translated to bulk creation and termination of threads running programs that implement the corresponding OpenCL operators. OpenCL's explicit memory types can be discarded as there is only one shared memory system.

Note that the existing languages' semantics cannot be translated to SL source text, in other words SL cannot serve as a common intermediate representation for the diversity of

concurrency constructs in existing languages. This is because these languages also contain additional data types, synchronization devices and scheduling hints which do not have equivalents in SL, *although they could be implemented by directly targeting the machine interface from chapter 4 in a dedicated code generator.*

To conclude, we do not advertise our SL extensions as the only possible interface language to the target architecture. Common interfaces, if any were defined, would likely exist as the internal intermediate representations in compilers' middle-ends. Instead, we propose our work as a modest yet practical vehicle to bootstrap further research in this environment.

6.4 Operating software services

We subsequently used the aforedescribed new compiler technology to port operating software code as described in the sections below.

6.4.1 Minimal run-time requirements

The minimal run-time environment required by the test and benchmark applications we considered includes support for:

- the following components from the standard C library: math functions, string manipulation, console output (`stdout`, `stderr`), heap allocation.
- the following POSIX services: abnormal asynchronous termination (`abort`), file I/O (`read`/`write`).

Autonomous services from the C library can be obtained cheaply by reusing standard F/OSS implementations, as we describe below in section 6.4.3. Existing heap allocators mostly require only the traditional service `sbrk` which can be implemented cheaply on a flat address space. In contrast, console and file access are a more costly requirement because they imply full-fledged support for external I/O channels, persistent storage and filesystems. To implement these cheaply, we used the heterogeneous integration with companion cores described in section 5.5.1.

6.4.2 Operating system services

□ We used companion processors to support file access: we implemented "proxy" functions for the base POSIX calls `open`, `close`, `read`, `write`, `link`, `unlink`, `sync`, `dup`, `dup2`, `getdtablesize`, `fsync`, `rename`, `mkdir`, `rmdir`, `stat`, `fstat`, `lstat`, `opendir`, `fdopendir`, `rewinddir`, `telldir`, `seekdir`, `closedir`, `readdir`. This was sufficient to implement the streams and file I/O APIs from the standard C library and support direct uses of the system interface for file access by programs. We also wrapped the supplementary X/Open "parallel I/O" operations `pread` and `pwrite`, because contrary to `read` / `write` these provide the file position explicitly and can thus be issued concurrently from separate threads without interference. We surmise that extended file access APIs such as POSIX's asynchronous I/O interfaces (`aio_read`, `aio_write`) can be added similarly at minimum cost.

□ Several system services, in contrast, did not require to delegate the behavior to the companion processor: system memory allocation (`sbrk` and anonymous mappings with `mmap`) can run locally on the microthreaded cores to support C's `malloc`. Time and date functions (`gettimeofday`, which support C's `time`) can access a RTC device directly. We also used the substrate ISA's time-stamp counters to support C's `clock`.

□ To implement console input-output (POSIX's special file descriptors 0, 1, 2), we implemented a configuration option which selects between various I/O device combinations (e.g. either the UART or matrix display devices).

▷ The following features were designed but are not yet implemented:

- for dynamic process creation (`vfork`+`exec`) and process data encapsulation, a loader can make a copy of the initial program data segments in the shared address space, a strategy suitable with an Alpha ISA substrate, which uses GP addressing, or alternatively the entire program image is duplicated and re-linked, as suggested with the default image format for the SPARC ISA substrate;
- the environment variables (for C's `getenv`) are loaded from ROM for the first program, and subsequently inherited upon further process creations.
- during program initialization, the loader creates a new thread of execution from the program's entry point, which maps to the C library initialization routine, which in turn initializes the C library's internal state (e.g. `malloc`'s allocation pool), runs any program-defined data constructors and finally transfers control to the program's `main` function.

Instead, in our implementation we provided a simple loader able to execute a single monolithic program with both application and operating software in a single memory image. We suggest the features above as future work.

6.4.3 C library services

To avoid re-implementing a C library from scratch, we studied how to reuse existing code. To select a code base, we pre-selected existing implementations satisfying the following requirements:

‹L1› *available in source form*, either in C or assembly source for one of the target ISAs, because it had to be recompiled/re-assembled to the binary format suitable for the new architecture;
‹L2› *suitable for cross-compilation*, since the platform where the code was compiled would not be the target architecture;
‹L3› *suitable for execution on a shared memory heavily multithreaded platform*, since this is the basic setting of all selected benchmarks;
‹L4› *modular*, since not all library functions would be needed and some could not be possibly supported initially anyways, such as support for file access;
‹L5› *compiler-agnostic*, since we could not guarantee support for specific existing non-standard C extensions beforehand.

We then analyzed those few codebases that satisfy requirements ‹L1› and ‹L2›: newlib [Red], μClibc [YMBYG08, pp. 115–127] [And], the GNU C library [Freb], the standard C library of BSD systems [MBKQ96, McK99], the standard C library of OpenIndiana[2] [JIS].

Our results are summarized in table 6.2. The implementations from GNU and OpenIndiana needed to be excluded early because of requirement ‹L5›. We then considered the following choice: either start from a thin embedded implementation and improve it to ensure that it would be suitable for use in a large multithreaded setting, or start from a larger

[2]OpenIndiana was forked from OpenSolaris when Oracle discontinued the OpenSolaris project.

Implementation	‹L3›	‹L4›	‹L5›
newlib	yes	no	yes
μClibc	no	yes	yes
BSD	yes	yes	yes
GNU	yes	no	no
OpenIndiana	yes	no	no

Table 6.2: Requirements satisfied by existing C library implementations.

The listed implementations all satisfy requirements ‹L1› and ‹L2›.

implementation that satisfied requirement ‹L4› and reduce its dependencies on underlying operating system support.

We adopted the latter approach, starting from the BSD components, for the following reasons:

- analyzing the uclibc and newlib implementations revealed multiple shared global-scope variables and a design made with the assumption of sequential execution; adding the necessary state protection to arbitrate state access during concurrent execution seemed an arduous task, with the associated high risk of errors that would add a potentially costly troubleshooting overhead to the benchmarking efforts;
- due to its longer history and wide audience, the BSD library is more comprehensive feature-wise and thus its selection reduced *a priori* the risk that our choice would fail to support any additional application requirement discovered in a later phase of our research.

□ By reusing components from the FreeBSD project [MNN04] with few changes, we were able to provide comprehensive support for the following C standard services: diagnostics (`assert.h`, [II99, B.1]/[II11b, B.1]); character handling (`ctype.h`, [II99, B.3]/[II11b, B.3]); errors (`errno.h`, [II99, B.4]); characteristics of floating types (`float.h`, [II99, B.6]); sizes of integer types (`limits.h`, [II99, B.9]/[II11b, B.9]); mathematics (`math.h`, [II99, B.11]/[II11b, B.11], support for `long double` omitted); variable arguments (`stdarg.h`, [II99, B.14]/[II11b, B.15]); boolean types and values (`stdbool.h`, [II99, B.15]/[II11b, B.17]); common definitions (`stddef.h`, [II99, B.16]); integer types (`stdint.h`, [II99, B.17]); buffered output functions, including formatted output, on standard output streams and character strings (`stdio.h`, [II99, B.18]/[II11b, B.20]); a subset of the general utilities (`stdlib.h`, [II99, B.19]/[II11b, B.21]); a subset of the string handling functions (`string.h`, [II99, B.20]/[II11b, B.23]); a subset of the date and time functions (`time.h`, [II99, B.22]/[II11b, B.26]).

▷ The remaining standard library services from [II99] and other newer features from [II11b] were not implemented because they were not needed by the test applications we considered. If they were ever needed, the following could be imported from existing codebases at little cost: extensions from [II11b] to the services already listed above, format conversions of integer types [II99, B.7]/[II11b, B.7], alternative spellings for operators [II99, B.8]/[II11b, B.8], localization [II99, B.10]/[II11b, B.10], alignment operators [II11b, B.14], standard input and file access [II99, B.18]/[II11b, B.20], the arithmetic general utilities [II99, B.19, `abs`, `div`, etc.]/[II11b, B.21], multi-byte/wide character utilities [II99, B.19,B.23,B.24]/[II11b, B.21,B.27,B.28,B.29], and type-generic math functions [II99, B.21]/[II11b, B.24], the "no return" function specifier from [II11b, B.22].

▷ In contrast, the following services would require more research and effort:

- support for complex arithmetic (`complex.h`, [II99, B.2]/[II11b, B.2]) and long floating types (`long double`) would require extra attention during code generation as they require more than one machine register (or synchronizer in the proposed architecture) for a single variable;
- support for controlling the floating point environment (`fenv.h`, [II99, B.5]/[II11b, B.5]) requires a hardware interface to control the FPU in the platform;
- non-local jumps (`setjmp.h`, [II99, B.12]/[II11b, B.2]) require platform-specific, hand-coded assembly code as they cannot be expressed directly in C;
- signal handling (`signal.h`, [II99, B.13]/[II11b, B.13]) conceptually conflicts with the purpose of hardware microthreading, a topic which we revisit later in sections 14.2 and 14.5;
- the process management interfaces (`system`, `atexit`, [II99, B.19]/[II11b, B.21]) require operating system support for separated processes, a topic which we revisit later in sections 14.4 and 14.5;
- support for atomic object access and threads introduced in [II11b, 5.1.2.4, B.16, B.25] does not map directly to the proposed concurrency management features, a topic which we revisit in chapter 7.

To summarize, the large compatibility of the proposed C compiler with legacy C code allowed us to reuse large existing code bases for operating software components. The resulting combination of compiler and library code provides a large subset of the standard *hosted* C execution environment from [II11b, 4§6, 5.1.2.2].

Summary

○□ We have acknowledged prior work which has attempted to provide an interface language to the proposed architecture, and we demonstrated the shortcomings of this approach. We have outlined the design requirements for a machine interface language derived from C. We then adopted a practical implementation strategy based on maximum reuse of existing tools. Using this strategy, we succeeded in providing a C compiler for the target architecture. We also provided a *meta-compiler* that implements transformations of a C language extension, which we call "SL," towards multiple platforms including the proposed architecture from part I. While our implementation exploits compiler features specific to GNU, we also show that the strategy is portable to other compiler substrates. Then using our freestanding programming environment, we subsequently ported existing operating software components
▷ suitable to run hosted C programs on the platform introduced in chapter 5. We also outlined how to extend this software support in future work.

Chapter 7

Disentangling memory and synchronization for shared state and communication

> The important thing in science is not so much to obtain new facts as to discover new ways of thinking about them.
>
> William L. Bragg

Abstract

In this chapter, we highlight the opportunity to synchronize program behavior by means other than memory. Then we show how to expose synchronization and consistency semantics in the proposed interface language from chapter 6.

Contents

Side note 7.1: About implicit communication.

From an information-theoretic perspective, communication exists as soon as information is exchanged between a sender and a recipient. Communication requires that the parties share a commonality *a priori*, usually shared semantics for the language of messages; however it typically does not require a third party to participate in the exchange. We recognize two extended forms of communication which actively rely on a third party. The first is *proxied communication*, where the third party acts as a relay for the signal without sharing the commonality between sender and recipient. Encrypted network channels are examples: the intermediary network nodes that carry the encrypted bits do not know the keys necessary to provide meaning to the messages exchanged. The second is *implicit communication*, where a third party *is* the shared commonality. For example, the meaning of the message "the meaning of this message is to be read from the first book published tomorrow" cannot be derived from the text of the message itself; instead it must be fetched from the external shared commonality, in this case the first book published tomorrow. In this chapter, we are considering implicit communication that occurs between processors using the information in shared storage devices as shared commonalities.

7.1 Introduction

Ever since the advent of architectures where multiple processors, or virtualizations thereof via threads, can access a single storage device via their interconnect, computer users at all levels of abstraction have attempted to exploit the opportunity for implicit communication which exists between stores issued by one processor and loads issued by another, *regardless of whether the processors may be connected to each other via dedicated communication channels.* To clarify, we provide our definition of "implicit communication" in side note 7.1.

While investigating diverse applications and languages, we were able to reduce all uses of implicit communication to only two essential patterns. The first is *persistent shared state* in multi-agent computations, e.g. the service provided by a shared file system. This is characterized by the ability to observe the state even when no agent is active, and the ability of the agents to access the state without knowing *a priori* which other agent last accessed it. The other is *implicit communication channels*, i.e. the ability to communicate an arbitrarily shaped data structure by using only a reference to it. We distinguish these two *services* here to ground the following observations:

- shared state and communication channels are abstract concepts which do not *require* a physically shared storage to be implemented. For example, if shared state can be represented logically, it can be transferred between the physical locations where it is used without the need for a central repository; this is the principle behind REpresentational State Transfer (REST) [Fie00]. Communication channels, in turn, can be implemented by serializing the arbitrarily shaped data over dedicated links between processors, separate from the memory system;
- each of these two concepts requires different kinds of support from the substrate hardware with regards to *synchronization.* To implement shared state, the substrate must provide mechanisms for *transactions*, to provide an illusion of atomicity of multiple updates by one agent from the perspective of other agents. To implement channels, the substrate must provide mechanisms to *guarantee order* of communication events and *signal* the availability of data or the readiness of the agents to participate in a communication activity.

Despite this distinction, the conceptual model of general-purpose processors has been historically simplified to the point that *the memory interface has become the only communication interface between the processor and its outside world.* There are three consequences to this:

- this simplification has allowed programming language designers to conflate the concept of *side effect* with the concept of *memory operation*, whereas previously side effects would also include actions on other physical links than those connecting the processors to memory (e.g. dedicated I/O links);
- the simplification has diluted the need in programming languages to *expose the shape and purpose of data* accessed concurrently by different processors to the underlying platform: all memory addresses are considered symmetrical with regards to their reachability, and programs do no longer explicitly distinguish which addresses participate in a multi-processor activity from those that don't;
- it has created a general understanding and expectation from software implementers that the memory system is "*the great coordinator*" between processors, and that the only synchronization primitives available in programs are one-size-fits-all memory-centric mechanisms like "atomic fetch-and-add" or "atomic compare and swap" that can be used to implement both shared state and communication channels.

This conflation of concepts is worrying. By erasing the conceptual nuance between state, communication and memory, it erases the fact that shared state and communication are actually built upon fundamentally separate synchronization mechanisms. The *innovation space* where architects could introduce other features, next to memory, with different and *more efficient* characteristics, is thereby reduced. Moreover, it creates pressure on the providers of memory services, i.e. memory architects and system integrators, to provide increasingly comprehensive mechanisms for synchronization. Mechanisms to control the ordering and signalling required by logical communication channels, in particular, increase the overall logic and energy cost of the memory system as a whole *even for those uses of memory not concerned by multi-processor interactions.* This pressure is the origin of discussions about "memory consistency", which we revisit below in section 7.1.1.

Hopefully, future architectural innovations will safeguard and exploit the conceptual nuances identified above in general-purpose computers. We can recognize an example step in this direction with "scratchpad memories," [BSL+02] where small, dedicated and non-synchronizing memories are placed next to cores on Multi-Processor Systems-on-Chip (MPSoCs) with other mechanisms like hardware mailboxes [FH76] to organize synchronization and communication between cores.

The proposed architectural innovation we described in part I also contributes to this vision. As we explain below in section 7.1.2, it negotiates synchronization between processors via mechanisms distinct from memory. We show a possible abstraction of these semantics in a system-level language interface in sections 7.2 and 7.3, which we apply in sections 7.4 and 7.5. We then discuss related work in sections 7.6 and 7.8.

7.1.1 The pressure on memory: from simple to less simple

The original definition of "memory" is a physical device which provides some guarantee that the values sent via store operations will be read by subsequent load operations. It is fundamental because it is one of the two halves of the general computer, the other half being the processor, which makes the computing system analogous to a Turing machine, and thus *able* to compute (cf. section 1.2.1). It is also quite simple and inflexible: the memory must guarantee that loads from one processor will return the value most recently stored by that processor, no matter what; otherwise, computation is not possible. This inflexibility holds for a single-threaded processor, or in a concurrent system for its simplest virtualization, the single thread.

Once multiple processors, or threads, are connected together, a choice then exists in a design: either each of them can only access its own memory isolated from other processors, or they can *share* storage devices. As outlined above, the *purpose* of such sharing is a desire of software implementors to enable *implicit communication* between stores issued by one processor and loads from another, in a quest for more simplicity in programming models. Without this sharing, processors can typically only communicate via explicit point-to-point messages ("message passing").

The first implementations were simple: each "memory" was a single physical device, and the links between processors and memory simply preserved the ordering of the operations as issued by the processors. The use of such an architecture in programs is not too complicated: stores by any processor are visible in the same order by all other processors sharing the same memory. This guarantee is called "*sequential consistency*," and was long the *de facto* machine abstraction for multi-programming.

What happened then was a back-pressure from hardware architects.

One force was the introduction of *caching*, especially separate caches for separate processors connected to the same backing store. As storage capacity grows, latency of access grows too. When its sharing factor increases (more processors per storage unit), contention occurs and the latency of access grows more. Caching is then desirable to provide the illusion of uniform, lower latency to access larger storage, given some locality of access. However, *cache protocols*, in particular *coherency protocols* responsible for propagating cache updates between multiple processors, are expensive to implement if they must provide the illusion of sequential consistency. It is much cheaper for the architect to "break the contract" in part, for example by saying that a store by a processor will only affect its local cache and that further loads by other processors will not see the update until the first processor explicitly flushes its changes. This is simpler to implement and incurs less pressure on the cache-memory network, but it does not respect the expectation of global visibility in the sequential model. As such, it makes the implementation of shared state and communication channels over memory, the two applications identified above, somewhat more difficult.

The other force was the introduction of *reordering* across network links, for example for congestion control or extra ILP within processors. Even in the absence of caches and using a single memory, reordering can cause stores from two processors to arrive in a different order than they were emitted. For example, a thread A can perform "a := 10; b := 20; barrier" and a thread B can perform "barrier; a := 30; b := 40" and the memory ends up containing "a=30" and "b=20". Again, this contradicts an expectation of the sequential model, this time the preservation of store order. This constitutes another obstacle to the implementation of shared state and communication channels.

In addition, both techniques (caching and reordering) can combine, creating even more exotic situations. The *status quo* at the time of this writing is a trade-off, where the architect combines the cheaper consistency protocols with extra control over the memory system, namely *barriers* to flush stores globally, *acquire-release locks* to delineate a subset of stores that must be implicitly propagated globally, and *reordering fences* which disable reordering selectively, usable by programs to declare their intent to obtain the illusion of sequential consistency for identified sequences of loads and stores. We should deem this state of affairs still unsatisfactory, as these primitives do not *scope* synchronization: when updates must be propagated, the memory system must assume that they must become visible *everywhere*. Even though application knowledge may exist to inform that the data is needed only in a specific area of the system, and thus create an opportunity to save synchronization latency

and energy, there is currently no mechanism to communicate this knowledge to the hardware in general-purpose designs.

To summarize, issues of consistency, which are at the heart of the discussion about the programmable semantics of multi-processor systems where memory coordinates all interactions, is the result of the collision between software implementers, who strive for conceptual simplicity, and hardware architects, who strive for simplicity in the implementation towards more efficiency and scalability.

7.1.2 Synchronization and memory in the proposed architecture

The innovation from part I is focused on the internal organization of cores for latency tolerance, and the efficient coordination of work distribution across multiple cores. It proposes that threads can synchronize using dedicated dataflow synchronizers in hardware within cores, and uses a dedicated delegation network combined with a distributed bulk creation and synchronization protocol implemented in hardware.

What interests us here is that this innovation is fundamentally agnostic of how the cores are connected to storage, i.e. it does not mandate a specific memory architecture nor does it propose specific mechanisms to coordinate access to data. Instead, it proposes the same synchronization mechanisms regardless of whether a shared memory system is available, and regardless of what consistency semantics are available in the shared memory, if any.

Of course, this approach does not *preclude* a chip design using a strongly consistent memory system in combination with the proposed core design and inter-core synchronization subsystem. For example, the FPGA-based implementation introduced in section 4.7 connects the proposed core design to a traditional memory bus. However, at the same time the approach strongly *suggests* exploring whether the hardware synchronization mechanisms, which are independent from memory, could provide a powerful multi-core programming environment when coupled with a *simple, efficient, scalable but weakly coherent* memory system that does not offer synchronization mechanisms.

This is the approach that we introduced previously in section 3.4.1. For example, the reference processor chip implementation that we used to define the platform from chapter 5 provides cache coherency between cores that share a single L2 cache, but updates to L2 caches are only propagated upon bulk creation and synchronization events. This means that a thread running on a core connected to an L2 cache cannot communicate or share state reliably *using memory* with a thread running on a core connected to another L2 cache, *even though they can organize communication and synchronization using the dedicated synchronization network*.

Given these circumstances, we explored whether we could abstract the semantics of our platform in an abstract machine suitable for a programming language like C. This process is especially important because most computing ecosystems rely on the abstract machine of system-level language interfaces to construct software and define its semantics, instead of using knowledge about the specific hardware platforms. The rest of this chapter documents our findings.

7.2 Extending C with memory-independent synchronization

□ In this section we explain how the ordering of concurrent operations is decided from the order of execution of special *synchronizing operations* by programs, independently from the order of loads and stores to memory.

7.2.1 Synchronizing operations and "scheduled before"

1 We consider the execution of programs on a parallel machine, where virtualized processors, i.e. threads, enter and leave the system dynamically.

2 Each thread executes a thread program consisting of *instructions*, each specifying the execution of one *operation* at run-time. The operations correspond to an observable effect on the direct environment of the processor that executes it, for example changing the state of an I/O device, advancing an instruction counter, or *issuing* a load or store to a memory system.

3 We call the program order between instructions, specified by the thread program's control flow in a programming language, the "*sequenced before*" order, and we denote it $\rightarrow$. This is equivalent to the relation of the same name in [II11b, 5.1.2.3§3].

4 Certain operations *synchronize* with other operations performed by another thread. This relation is asymmetric: if b synchronizes with a, the execution of b does not start before a completes, but no extra information is given as to when the execution of a starts.

5 We name the partial ordering of the execution of operations at run-time "*scheduled before*", we denote it $\rightsquigarrow$, and we specify:

- "sequenced before" constrains "scheduled before": if a and b are instructions, and a_r and b_r are the corresponding operations at run time, then $a \rightarrow b \Rightarrow a_r \rightsquigarrow b_r$
- all implementations of the abstract machine guarantee that the execution of a ready operation x starts a finite amount of time after all operations $\{y \mid y \rightsquigarrow x\}$ have completed. This guarantees progress of execution for all schedulable operations.

6 We say that two operations a and b are *concurrent* if $a \not\rightsquigarrow b$ and $b \not\rightsquigarrow a$. An implementation *may* execute concurrent operations simultaneously, or choose an arbitrary scheduling order between them.

7 The definition of the special synchronizing operations below extends the $\rightsquigarrow$ relation and thus constrains scheduling further.

7.2.2 Threads and families

7.2.2.1 Threads

1 We denote the set of all logical threads $\mathcal{T}$.

2 For readability, we will denote henceforth "$x^{(i)}$" for "operation x executed by thread $i \in \mathcal{T}$," and $\overset{i}{\rightarrow}$ the sequence order of operations executed by i.

3 We denote $begin(i)$ and $end(i)$ the minimum and maximum of $\overset{i}{\rightarrow}$, i.e. the first and last operations executed by a thread. Threads that perform no operations can be considered to execute a single pseudo-operation with no effect so that *begin* and *end* are always defined.

7.2.2.2 Families

1 $\mathcal{T}$ is partitioned in totally ordered subsets named *families*.

(the grouping of threads into families are an emergent sub-structure of $\mathcal{T}$ caused by the use of the c operation (described below in section 7.2.3) by programs)

2 We denote $F(i) \subset \mathcal{T}$ the family of thread i, and $\prec$ the total order within a family.

3 We denote:

- for any family F, the *first thread of F*, denoted *first(F)*, that dominates all other threads in F via $\prec$;

- for any family F, the *last thread of F*, denoted *last(F)*, which is dominated by all other threads in F via $\prec$;
- for any family F and $t \neq last(F) \in F$, the *successor of t* in F, denoted *succ(t)*, verifying $\nexists\, t_2 \in F : t \prec t_2 \prec succ(t)$.

(the last thread in a family has no successor; in a family of one thread, the only thread is both the first and last thread)

4 Each family F is further partitioned as a set of *sequential segments* $seq(F)$, which are sub-sets of participating threads that are executed internally sequentially.

5 We further name *segment prefix* the sub-set of a sequential segment that contains all threads in that segment but the last via $\prec$, i.e. $\forall S \in seq(F), \quad prefix(S) = \{t \mid t \in S \wedge \exists t' \in S : t \prec t'\}$.

6 Sequential segments further constrain scheduling as follows:

$$\forall S \in seq(F), \forall t \in prefix(S), \quad end(t) \rightsquigarrow begin(succ(t))$$

Rationale: This denotes that within a sequential segment, the end of a thread dominates the start of its successor in the scheduling order. This expresses that there is no concurrency between threads that belong to the same segment, whereas concurrency may still exist across separate segments. This constraint reflects the sequential scheduling of logical threads over thread contexts introduced in sections 3.3.1 and 4.2.

7.2.3 Schedule ordering of threads from family creation

1 The abstract machine provides a special synchronizing operation $c[F]$, called "family creation."

2 This constrains scheduling as follows:

$$\forall t \in F \quad c[F] \rightsquigarrow begin(t)$$

7.2.4 Schedule ordering of threads upon family termination

1 The abstract machine provides a special synchronizing operation $s[F]$, called "family synchronization."

2 This constrains scheduling as follows:

$$\forall t \in F \quad end(t) \rightsquigarrow s[F]$$

7.2.5 Dataflow synchronizers

1 The abstract machine defines a set $\mathcal{O}$ of *dataflow synchronizers* and a set $\mathcal{V}$ of *unit values*.

2 It then provides the following special synchronizing operations on these synchronizers, denoted for any $o \in \mathcal{O}$:

- $r_s(o)$ for "start synchronizing read,"
- $r_f(o)$ for "finish synchronizing read,"
- $q(o)$ for "query synchronizer,"
- $w(o, v)$ for "synchronizing write" ($v \in \mathcal{V}$), and
- $e(o)$ for "clear synchronizer."

3 "Start synchronizing read" and "finish synchronizing read" are pseudo-operations that always come in pairs in programs, expressed as a single concrete "synchronizing read" instruction that entails both operations immediately one after the other during execution.

Rationale: We distinguish them because only the *completion* of a synchronizing read is constrained, as described below.

4 We will denote "$xs(o)$" for "any operation that is either a r_s, r_f, q, w or e operation on o."

5 All xs operations on a given synchronizer are atomic, even across threads:
$\forall o, \forall xs(o), \forall xs'(o) \neq xs \quad xs \rightsquigarrow xs' \vee xs' \rightsquigarrow xs$

6 "Finish read" operations synchronize with writes the first time after a clear:
$\forall e(o),\ \forall r_f(o) : e \rightsquigarrow r_f \wedge (\nexists\, e' : e \rightsquigarrow e' \rightsquigarrow r_f),\ \forall w(o,v) : e \rightsquigarrow w \wedge (\nexists\, w' : e \rightsquigarrow w' \rightsquigarrow w) \wedge (\nexists\, e' : e \rightsquigarrow e' \rightsquigarrow w),\ w \rightsquigarrow r_f$

7 The effect of executing an unprovisioned "finish read" operation is undefined, i.e. any r_f such that $e \rightsquigarrow r_f \wedge (\nexists\, w : e \rightsquigarrow w \rightsquigarrow r_f)$ may not complete, trigger a fault to signal deadlock or effect some other unspecified behavior.

8 Each read $r_f(o)$ evaluates to the value $v \in \mathcal{V}$ stored by the most recent $w(o,v)$ according to $\rightsquigarrow$: $\forall r(o), \forall w(o,v) \qquad (\nexists\, e(o) : w \rightsquigarrow e \rightsquigarrow r) \wedge (\nexists\, w' : w \rightsquigarrow w' \rightsquigarrow r) \Rightarrow r(o) \quad \text{yields} \quad v$

9 Each query $q(o)$ evaluates to the value $v \in \mathcal{V}$ stored by the most recent $w(o,v)$ according to $\rightsquigarrow$, *if any exists*; otherwise, i.e. if there is no last w or if the last non-q operation on o is a r_s or e operation, it evaluates to a non-deterministic, unspecified value of $\mathcal{V}$.

10 Each thread i is associated with a partial function of $\mathbb{Z}$ to $\mathcal{O}$, denoted $V[i]$, that represents its visible dataflow synchronizers. (For example $V[i](n) \in \mathcal{O}$ is the n-th synchronizer visible by i.)

7.2.6 Mapping of synchronizers to families

1 $\mathcal{O}$ is partitioned between families, i.e. a given synchronizer o may be visible from multiple threads within a family but is not visible from threads belonging to different families:
$\forall (i, n, i', n') \quad F(i) \neq F(i') \Rightarrow V[i](n) \neq V[i'](n')$

2 The execution of a program shall occur as if an initial "clear" operation was issued to every visible synchronizer prior to the execution of all threads, i.e. $\forall i \in \mathcal{T}, \forall o \in V[i], \exists e_0(o) : e_0 \rightsquigarrow \mathit{begin}(i)$

7.2.7 Dataflow synchronization between families

1 The abstract machine provides two operations $\bar{w}[i](n,v)$ $(i \in \mathcal{T}, n \in \mathbb{Z}, v \in \mathcal{V})$ and $\bar{q}[i](n)$, respectively for "remote synchronizing write" and "remote query," which may be executed by other threads than i.

2 The execution of one $\bar{w}[i](n,v)$ operation incurs, after a finite amount of time, the execution of one $w(V[i](n), v)$.

3 The execution of one instance of $\bar{q}[i](n)$ incurs, after a finite amount of time, the execution of one $q(V[i](n))$; the execution of $\bar{q}$ further produces in the thread where it is issued the value produced by the corresponding q.

7.2.8 Mapping in the language interface

The items defined above are exposed in the proposed SL extensions to C as follows:

- C's evaluations map to operations of the abstract machine;

- thread programs in SL, and the C functions called recursively from them, map to thread programs in the abstract machine;
- the "`sl_create`...`sl_sync`" construct entails the execution of a c (family creation) operation no earlier than the point "`sl_create`" is reached during execution, and no later than the point "`sl_sync`" is reached;
- the "`sl_create`...`sl_sync`" construct entails the execution of a s (family synchronization) operation at the point "`sl_sync`" is reached during execution, and thus execution does not proceed past "`sl_sync`" until all threads in the created family have terminated;
- the logical index range and "block size" parameters to "`sl_create`" define the sequential segments of the created family, as detailed in Appendix I.5.8.1;
- SL's dataflow channels map to the abstract machine's dataflow synchronizers;
- the "`sl_seta`" construct entails the execution of a $\bar{w}$ operation, and "`sl_geta`" entails the execution of a $\bar{q}$ operation;
- the "`sl_setp`" construct entails the execution of a w operation, and "`sl_getp`" entails the execution of a r_s/r_f operation pair.

The q and e operations are intendedly not exposed in SL, although the platform implementation may support them, in order to encourage the expression of deterministic, sequentializable programs.

7.3 Extending C with multi-thread consistency

□ In this section, we explain under which circumstances stores by one thread are visible to loads by another thread, as a function of the scheduling order defined in the previous section.

7.3.1 General condition for consistency

1 The abstract machine provides a set $\mathcal{L}$ of *locations*, corresponding to "memory addresses."
2 It then defines two operations $ld(l \in \mathcal{L})$ and $st(l \in \mathcal{L}, v \in \mathcal{V})$, for "load" and "store" respectively.
3 It then defines for every location $l \in \mathcal{L}$, a partial *consistency order for* l, denoted $\overset{l}{\prec}$, over all ld and st operations operating on l.

Rationale: The partial order $\prec$ is defined independently for every memory address. This is intended to support the independent ordering of memory operations touching separate cache lines in a cache coherency protocol.

4 The abstract machine then defines a *visibility* property over ld and st operations as follows. Given some operations $st(l, v)$ and $ld(l)$, if the following conditions all hold:

- $st \overset{l}{\prec} ld$, and
  (a store st to a given location precedes a load ld from the same location)
- $\nexists\, st' \neq st \quad st \overset{l}{\prec} st' \overset{l}{\prec} ld$, and
  (there is no other store st' to the location that precedes ld and is preceded by st)
- $\nexists\, st' \neq st \quad st' \overset{l}{\not\prec} ld \wedge ld \overset{l}{\not\prec} st'$,
  (there is no other store st' to the same location that neither precedes ld nor is preceded by ld)

then st is *visible* from ld.

Rationale: This property establishes visibility as a function of the consistency ordering, separately for every memory address.
Note that the third condition is possibly non-intuitive: unless $\prec$ is further constrained, any store not related to a load via $\prec$ effectively "hides" any other store to the same location from that load, even those related via $\prec$ and including those dominated by the load. This provision exists because we consider weakly coherent cache systems which do not protect against erroneous race conditions in programs.

5 The abstract machine then defines a *provision* property as follows: any load ld is *provisioned* if there exists at least one store st visible from ld in $\prec$.

6 Further, the following hold:

- if an st operation starts, then it completes within a finite amount of time;

  Rationale: Stores should complete eventually.

- if a provisioned ld operation starts, then it completes within a finite amount of time; and the actual value produced by its execution is an element of the set of values written by the st operations to the same location that are visible from ld.

  Rationale: Loads should complete eventually if they are provisioned. Furthermore, if they are provisioned they yield one of the values stored most "recently" according to $\prec$.
  Meanwhile, the behavior of unprovisioned ld operations is undefined. For example, an implementation may cause halting, a deadlock, or produce a value that causes halting or deadlock of any subsequent operation using it, or produce a non-deterministically chosen valid value.

7 A program is *consistent* if all its possible executions that are compatible with the abstract machine guarantee that all loads are provisioned.

8 A load operation is a *data race* if it is provisioned by more than one store.

9 A program is *deterministically consistent* if it is consistent and executes no data race.

7.3.2 Communication domains

□

1 The abstract machine provides the notion of *Implicit Communication Domain (ICD)*.

2 Each pair of (thread, location) is associated with exactly one ICD, noted $C(i,l)$.

3 The abstract machine then specifies that precedence implies visibility, within the same communication domain:

$$\forall (i,j), \forall x^{(i)}(l), \forall y^{(j)}(l) \quad [C(i,l) = C(j,l) \wedge x \rightsquigarrow y] \Rightarrow x \overset{l}{\prec} y$$

Rationale: ICDs capture the notion of implicit communication between stores and loads *within a region of the system*, and *constrained by the partial ordering between threads*. In particular stores from different threads that are not mutually ordered via $\rightsquigarrow$ can "hide" each other as per clause 7.3.1§4. ICD boundaries correspond to boundaries in the system where implicit communication is not guaranteed, even between operations ordered via $\rightsquigarrow$.

7.3.3 Consistency domains

□

1 The abstract machine provides the notion of *Consistency Domain (CD)*.

2 Each pair of (thread, location) is associated with exactly one CD, noted $\bar{C}(i,l)$.
3 The abstract machine then specifies that all loads and stores within the same CD to the same location appear in some order globally visible within the CD:

$$\forall (i,j), \forall x^{(i)}(l), \forall y^{(j)}(l) \quad \bar{C}(i,l) = \bar{C}(j,l) \Rightarrow x \overset{l}{\prec} y \vee y \overset{l}{\prec} x$$

Rationale: CDs capture the notion of "eventual visibility" of stores for all subsequent loads *within a region of the system*. Stores not synchronized via $\rightsquigarrow$ cannot "hide" each other within a CD if there are no unordered stores issued outside the CD. CD boundaries correspond to boundaries in the system across which stores may not be propagated automatically.

4 Henceforth the set of all consistency domains is noted $\mathcal{C}$. We will also denote "x_C" for "an operation x executed by a thread associated to consistency domain C."

7.3.4 Memory communicators

□ With the definitions so far, threads can communicate arbitrarily via memory within CDs, and the store-load visibility follows the edges of the scheduling order within ICDs. However, no provision is made to make stores executed in one ICD visible to loads executed in another ICD. For this purpose, we introduce semi-explicit communication operations as follows.

1 The abstract machine defines a set $\mathcal{M}$ of *memory communicators*, and a set $\mathcal{R}$ of *relative locations* for use with communicators.
2 It then provides three *communicating operations*:

- $b[R](m)$, for "*bind* m to a set R of relative locations" ($m \in \mathcal{M}, R \subset \mathcal{R}$);
- $p(m)$, for "*propagate* updates to the locations bound to m" ($m \in \mathcal{M}$);
- $a(m)$, for "*activate* updates to the locations bound to m" ($m \in \mathcal{M}$).

(b and p are intended for use by the writer side, whereas a is intended for use by the reader side)

3 It also provides a *relative addressing* partial function t which translates, within a given consistency domain and relative to a given memory communicator, a relative address usable in programs into an address suitable for implicit communication through the communicator: $t : \mathcal{C} \times \mathcal{M} \times \mathcal{R} \mapsto \mathcal{L}$.
4 The abstract machine then proposes the following service: any store to a location that is scheduled after a b operation and before a p operation and which operates on a location bound by b, becomes visible to loads scheduled after an a operation if a is scheduled after p, given that b, p and a operate on the same communicator. In other words:

For any given communicator m, $\forall m \in \mathcal{M}$,
given two ICDs C_1 and C_2, $\forall (C_1, C_2) \in \mathcal{C}^2$,
given a range of relative addresses R, $\forall R \subset \mathcal{R}$,
for any b, p operations executed in C_1 and a executed in C_2 such that b is scheduled before p and p is scheduled before a, $\forall (b_{C_1}[R](m), p_{C_1}(m), a_{C_2}(m)) : b \rightsquigarrow p \rightsquigarrow a$,
for any relative address l in that range, $\forall l \in \mathcal{R}$,
if $l_1 = t(C_1, m, l)$ and $l_2 = t(C_2, m, l)$ then
for any st operation on l_1 scheduled between b and p in C_1, $\forall v \in \mathcal{V}, \forall st_{C_1}(l_1, v) : b \rightsquigarrow st \rightsquigarrow p$,
and for any ld operation on l_2 scheduled after a in C_2, $\forall ld_{C_2}(l_2) : a \rightsquigarrow ld$,

the execution of $ld(l_2)$ occurs as if an operation $st(l_2, v)$ was visible from it in the same ICD, i.e.

$$\exists st_{C_2}(l_2, v) : st \overset{l_t}{\prec} ld$$

(the $b(m)$ operation "captures" a range of addresses into m and starts "recording" stores, until a $p(m)$ operation which "propagates" the changes. The a operation in a different consistency domain then "activates" the changes so they can be used by further loads.)

If multiple stores to the same location occur between a b and p operation pair, or if multiple p operations are scheduled before a corresponding a, then only the last in the $\prec$ order will be visible to loads occurring after a.

7.3.5 Possible implementations of t, b, p and a

Distinct platform implementations may provide the b, p and a operation and t function via separate mechanisms. We provide three examples below.

Note that a, b and p are not synchronizing; therefore they must be combined with the other synchronization mechanisms from section 7.2 to ensure e.g. that a is not executed before p completes.

7.3.5.1 Using message passing and a "push" protocol

- b prepares output buffers for a communicator;
- t accesses the output buffers;
- p sends the buffer to the "destination" of the communicator, the data is received asynchronously at the destination;
- a finishes waiting on reception of the data.

7.3.5.2 Using message passing and a "pull" protocol

- b prepares output buffers and associates addresses with the communicator;
- p is no-op;
- a remotely reads the data from the "sources" of the communicator;
- on the origin side, t is the identity; on the pulling side, t either is the identity (single address space) or addresses a receive buffer.

7.3.5.3 Using the proposed platform

- the b and m operations are no-ops,
- t is the identity;
- p is implemented using a write memory barrier which flushes outstanding stores globally.

7.3.6 Mapping in the language interface

The items defined above are exposed in the proposed SL extensions to C by stating that the addresses of bytes in C objects map to locations in the abstract machine, and that accesses to objects in programs map to *ld* and *st* operations. The visibility of object updates from object reads is then decided by the consistency rules and the scheduling order as per the specification above.

```
1 thread i:                    thread j:
2       ...                        ...
3 A:    st(l1, v1)         D:    st(l2, v3)
4       ...                        ...
5 B:    st(l1, v2)         E:    ld(l2)
6       ...                        ...
7 C:    ld(l1)
8       ...
```

Listing 7.1: Two unrelated threads.

Note that no further language support is available here to control the communication operations b, p and a and use the t function. Because of this, the language described in Appendix I can only be used to program within one communication domain.

Extra SL support for cross-ICD communication has been explored by peers and is reported on in [Mat10]: their work proposes to introduce explicit "memory objects" into the language, which are a combination of dataflow synchronizers and memory communicators, operations on these that control b, p and a, and accessors for the t function. These are then automatically removed, at compile time or run-time, if both sides of a communicating activity are known to be running within the same consistency domain.

▷ We also sought to support cross-ICD communication in our proposed platform from chapter 5 without inserting new language constructs. We noted that a, b and t are transparent (cf. section 7.3.5.3 above), and then we proposed to insert a p operation in the generated code immediately before every c and s operation. This enables implicit cross-ICD communication from all cores. While we implemented this feature in our proposed compiler from chapter 6, we highlight that this technique incurs excess memory barriers when threads are created within the same ICD. A more efficient approach should thus attempt to determine automatically when p is not needed based on placement information and elide it in those cases.

7.4 Examples of incorrect synchronization

7.4.1 Abstract example to illustrate the hiding effect

Consider a program causing the concurrent execution of the two threads in listing 7.1, unrelated through synchronization. By construction, we have A $\overset{l1}{\prec}$ B $\overset{l1}{\prec}$ C and B is visible from C. Likewise, D $\overset{l2}{\prec}$ E and D is visible from E. The program is consistent.

Consider now a program executing concurrently the threads in listing 7.2. Because we do not know that C $\overset{l}{\prec}$ D, nor do we know that D $\overset{l}{\prec}$ C, we must understand that D hides both A and B, so C is not dominated by a store to l and the program is not consistent.

7.4.2 Invalidated idioms

We assume the following examples run within one CD.

A load *ld* may observe the value written by a store *st* that happens concurrently with *ld*. Even if this occurs, it does not imply that loads happening after *ld* will observe stores

```
1 thread i:                    thread j:
2       ...                        ...
3 A:    st(l, v1)          D:   st(l, v3)
4       ...                        ...
5 B:    st(l, v2)
6       ...
7 C:    ld(l)
8       ...
```

Listing 7.2: Two unrelated threads with a race condition.

```
1  int a, b;
2  sl_def(f) {
3    a = 1; b = 2;
4  } sl_enddef
5
6  sl_def(g) {
7    print(b); print(a);
8  } sl_enddef
9
10 sl_def(main) {
11  sl_create(,,,,,,,f);
12   sl_create(,,,,,,g);
13   sl_sync();
14  sl_sync();
15 } sl_enddef
```

Listing 7.3: Independent ordering of loads/stores to different addresses.

to another address that happened before *st*. For example, in the program from listing 7.3, it can happen that `g` prints 2 and then 0.

Double-checked locking is an attempt to avoid the overhead of synchronization. For example, the `twoprint` program might be incorrectly written as in listing 7.4: there is no guarantee that, in `doprint`, observing the store to `done` implies observing the store to `a`. This version can unexpectedly, but correctly print an empty string instead of `"hello, world"`. Instead, proper synchronization is achieved using listing 7.5.

Another incorrect idiom is busy waiting for a value, as in listing 7.6. As before, there is no guarantee that, in `main`, observing the store to `done` implies observing the store to `a`, so this program could print an empty string too. Moreover, if the threads are merely in the same ICD but maybe in different CDs, there is no guarantee that the store to `done` will ever be observed by `main`, since there is no synchronization between the two threads. The loop in `main` is not guaranteed to finish. There are subtler variants on this theme, such as the program in listing 7.7. Even if `main` observes `g != NULL` and exits its loop, there is no guarantee that it will observe the initialized value for `g->msg`.

```
1  char* a; bool done;
2  sl_def(setup) {
3     a = "hello, world";
4     done = true;
5  } sl_enddef
6  sl_def(doprint) {
7    if (!done) {
8      sl_create(,,,,,,,setup);
9      sl_sync();
10   }
11   print(a);
12 } sl_enddef
13
14 sl_def(twoprint) {
15  sl_create(,,,,,,,doprint);
16   sl_create(,,,,,,,doprint);
17   sl_sync();
18  sl_sync();
19 }
```

Listing 7.4: Implementation of `twoprint`, insufficiently synchronized.

```
1 sl_def(twoprint) {
2  sl_create(,,,,,,,doprint); sl_sync();
3  sl_create(,,,,,,,doprint); sl_sync();
4 } sl_enddef
```

Listing 7.5: Proper synchronization for `twoprint`.

```
1  char* a; bool done;
2  sl_def(setup)
3  { a = "hello, world"; done = true; }
4  sl_enddef
5
6  sl_def(main) {
7    sl_create(,,,,,,, setup);
8    while (!done) {}
9    print(a);
10   sl_sync();
11 } sl_enddef
```

Listing 7.6: Invalid busy waiting for a value.

```
1  typedef struct {
2    char *msg;
3  } T;
4
5  T* g;
6
7  sl_def(setup) {
8    T* t = malloc(sizeof(T));
9    t->msg = "hello, world";
10   g = t;
11 } sl_enddef
12
13 sl_def(main) {
14   sl_create(,,,,,,,, setup);
15   while (g == NULL) {}
16   print(g->msg);
17   sl_sync();
18 } sl_enddef
```

Listing 7.7: Invalid busy waiting on a pointer.

7.5 Pitfall: atomicity anomalies

Atomicity anomalies can appear when the *granularity of consistency* offered by an implementation is different from the one assumed by the high-level description of a program. Intuitively, the granularity of consistency corresponds to the minimum "size" of two values so that, if they are stored "next to each other" they will still have different locations.

The first anomaly is the following. Suppose that machine addresses identify 4-byte long objects, but the programmer is treating one machine object as 4 different 1-byte abstract objects each with a different (abstract) location. Two independent threads may each perform an update to two different abstract locations, expecting each update to be visible to a successor load within the same thread from the same (abstract) location. But, if these 1-byte values are packed into the same 4-byte platform location, then these stores are really concurrent stores to the same location. Consequently, one of the two stores may non-deterministically mask the other, or they may mask each other, and the update to one of the bytes, or both, may be lost. The solution to avoid this problem is to avoid packing together abstract objects that might be updated by concurrent stores. In our platform from part I and chapter 5, the machine granularity is 1 byte so this problem is avoided.

The other anomaly is the following. Suppose the system supports 4-byte concrete objects, but the programmer wants to manipulate an 8-byte logical object. If two concurrent threads each update the entire 8-byte object, the programmer might expect a common successor 8-byte load via $\rightsquigarrow$ to receive one of the two 8-byte values written previously. However, the 8-byte load may non-deterministically receive 4 bytes of one value and 4 bytes of the other value, because the 8-byte load is really two 4-byte loads, and the consistency of the two halves is maintained separately. Note that this problem can only occur if the load is a data race. When programs are written to avoid data races entirely, programmers need not worry about it. In our platform from part I and chapter 5, this granularity anomaly only appears with assignments to aggregate types whose size is larger than the underlying ISA's word size, that is 8 bytes on Alpha or 4 bytes on SPARC.

7.6 Relationship with the latest C and C++ specifications

Concurrently to our own research, the American National Standards Institute (ANSI) published a new specification for both the C and the C++ languages [II11b, II11a]. Compared to [II99, II03], this new version introduces *concurrency semantics* in both languages, purposely crafted by their respective working groups to be mostly compatible with each other. These additions can be summarized as follows:

- the notion of concurrently executing *threads* is introduced. C++ specifies that all threads are fairly scheduled ("Implementations should ensure that all unblocked threads eventually make progress." [II11a, 1.10§2]), whereas C does not.
- synchronization between threads is introduced via special *atomic* objects, which are regular memory-based objects declared with the specifier `_Atomic`. Accesses to the same atomic object are globally ordered, and special "acquire" operations (used for mutex locks) synchronize with special "release" operations (used for unlocks).
- the visibility of updates to non-atomic objects relative to subsequent evaluations of those objects is decided via a "happens before" partial order between operations, constrained between concurrent threads by the interleaving of accesses to atomic objects.

In short, the designers of these extensions assume that the memory is "the great coordinator" as we explained in section 7.1. Their definition of the "happens before" relation between operations is particularly complex, because it must simultaneously specify the visibility of non-atomic object updates and the visibility of updates to atomic objects. Although these updates have quite separate consistency rules (atomic accesses are synchronizing, non-atomic accesses are not), because they are both memory objects their definition must be interleaved to decide what constitutes a race condition vs. a consistent access.

In contrast, our semantics allow the programmer to reason about execution order independently from memory accesses. The resulting situation is that although we started from the same language (C and C++ as per [II99, II03]), our proposed semantics diverge from the direction taken by ANSI for C and C++. The question thus arises of whether this situation is desirable and what are its consequences.

Here we start by observing that the concurrency semantics of [II11b, II11a] can be emulated in our environment by containing the execution of the entire program within one consistency domain, and requiring the memory system to provide support for atomic transactions. This is possible because:

- neither [II11b] nor [II11a] mandates a minimum number of simultaneously executing threads, so the platform restrictions on the number of thread contexts may freely constrain how many of [II11b, II11a]'s thread creations may occur;
- all valid non-racy executions according to [II11b, II11a] are valid non-racy executions in our abstract machine within one CD;
- data races are defined in both [II11b, II11a] and our abstract machine to result in undefined behavior.

▷ Because [II11b, II11a] can be emulated, we can provide a *backward compatibility* environment in an implementation of SL, where any program that requires [II11b, II11a]'s semantics is forced to run on a region of the system contained within one CD, without changing the abstract machine. We conclude that our proposed abstract machine is more general than those envisioned by ANSI for C and C++. We then revisit how operating software can constrain the placement of programs to specific consistency domains in chapter 11.

7.7 Position of the model from the programmer's perspective

In [KMZS08] the author attempts to isolate criteria to classify programming models for parallel systems. The proposed criteria are:

1. *System architecture*: *shared memory* vs. *distributed memory.*
2. *Programming methodologies*: how concurrency is exposed to programmers (API, directives, language extensions, etc)
3. *Worker management*: how execution units are managed, *implicit* (MPI, OpenMP) vs. *explicit* (Pthreads)
4. *Workload partitioning scheme*: how workloads are divided in chunks (tasks), *implicit* (OpenMP) vs. *explicit* (MPI)
5. *Task-to-worker mapping*: how tasks are mapped to workers, *implicit* (OpenMP) vs *explicit* (POSIX threads)
6. *Synchronization*: time order in which shared data is accessed, *implicit* (UPC) vs. *explicit* (MPI)
7. *Communication model*: *private shared address space*, *message passing*, *global partitioned address space*, etc.

Within this classification we place our system as follows. With regards to system architectures, our system aims to target both shared memory systems and distributed memory systems. Information about locality, if available, can be used to optimize code and reduce communication overhead by eliding uses of the b, p and a primitives. The proposed abstract machine does not mandate a specific programming methodology. Our SL implementation (including the extensions presented by [Mat10]) use language extensions while another [vTK11] uses an API. Workload partitioning is explicit, but fine grained. Our system suggests the definition of separate threads even for very small units of work. Once this is done, the granularity can be made coarse again by aggregating threads together automatically (run as loops), as we suggest in chapter 10. This has been described also in [Mat10]. Worker management, task-to-worker mapping and synchronization are mostly implicit in our system. A form of optional explicit mapping is introduced later in chapter 11. The communication model, subject of this chapter, is a global address space augmented with *consistency and implicit communication domains*, that defines regions of the address space where communication can be largely implicit. Across domains, communication must be more explicit.

7.8 Other related work

Our abstract machine borrows concepts both from Global Address Space (GAS)-derived models such as UPC [UPC05], Fortress [ACH+08], Titanium [YSP+98], Co-Array Fortran [NR98], and from dataflow (implicit synchronization) models such as Cilk [BJK+95] and Google Go [Goo].

7.8.1 Weak consistency within ICDs, overview from Go

Go is a concurrent programming language developed by Google. It provides *goroutines*, which are small lightweight threads. Memory consistency between concurrent goroutines is relaxed.

The weak consistency model proposed by our system within a given implicit communication domain shares much similarity with Go. The similarity is so close that the description of Go's memory consistency[1] also describes consistency within an ICD in our system. In particular, by stating that stores not ordered by thread synchronization can hide each other, Go shares our visibility semantics from clause 7.3.1§4.

7.8.2 Comparison with Cilk

Cilk [BL93, BJK+95, Joe96] was developed by MIT as both a programming language and an execution model. The Cilk programming language extends a subset of C in a way that a deterministic Cilk program stripped of all Cilk keywords becomes a valid C program with the same functional behavior.

However the Cilk execution model is more general than the Cilk language itself. The unit of work in Cilk, as with our system, is a logical thread. Thread definitions in Cilk are also schedule agnostic, i.e. the synchronizing dependencies between threads are explicit and the run-time system only guarantees that they are satisfied without specifying the execution order.

The essential difference between Cilk and our system is the granularity of synchronization. In Cilk, a thread begins to execute only after *all* its dependencies are satisfied; it then runs *until completion* at which point its output is checked to see if it satisfies a dependency for another thread. In our system, threads can start even when not all their dependencies are satisfied; running threads can wait for termination of threads they have created previously; and ordering is guaranteed between individual reads and writes to *dataflow synchronizers* visible across threads. Also, new threads can be created and start executing even during the execution of their parents.

Due to this difference, our execution model is more general and fine-grained than Cilk with respect to synchronization and concurrency. This stems from the following observation: any Cilk program using N concurrent execution units can be executed within our model with N execution units, using only one consistency domain. However a program that can use N concurrent execution units in our model may not be able to use more than M units when expressed using Cilk primitives, with $M < N$, because parent/child and family siblings which are concurrent in our system would need to be scheduled in sequence in Cilk to satisfy dependencies.

Apart from this principal difference, the consistency models proposed for Cilk ([Joe96, Chap. 6], [BFJ+96]) and our model are similar. In Cilk, stores are visible from loads if they dominate loads according to the directed acyclic synchronization graph defined by inter-thread dependencies. This is called "DAG consistency". In our model, stores are also visible from loads if they dominate loads according to the directed acyclic precedence graph with $\rightsquigarrow$ edges. In both models, multiple stores visible from a single load are resolved non-deterministically. Our model further states that consistency is not decidable when there exists a store that is not ordered with a load, whereas Cilk leaves this topic unspecified.

Also, our system extends the Cilk model by defining communication across consistency domains. This extension preserves the simplicity of defining consistency based on the dependency graph, but requires to enclose reads and writes to memory between a, b and p communication operations.

[1] `http://golang.org/doc/go_mem.html`

7.8.3 Communication granularity, comparison with CAF / UPC

In [CDMC+05], the authors study Co-Array Fortran (CAF) and Unified Parallel C (UPC) as two languages that offer SPMD over a global address space. Their primary statement is that:

> [...] CAF and UPC programs deliver scalable performance on clusters only when written to use bulk communication. However, our experiments uncovered some significant performance bottlenecks of UPC codes on all platforms. We account for the root causes limiting UPC performance such as the synchronization model, the communication efficiency of strided data, and source-to-source translation issues. We show that they can be remedied with language extensions, new synchronization constructs, and, finally, adequate optimizations by the back-end C compilers.

The solution the authors propose to optimize communication is twofold. The first is to adapt the granularity of synchronization to the actual functional requirement of programs, i.e. minimize the scope of synchronization and avoid barriers. This is present in our system already since all synchronization is explicitly scoped by default.

The other solution is to group individual accesses to memory into bulk communication of entire chunks of memory. Then they suggest directions to partly automate this process and synthesize bulk transfers:

> We believe that developing effective compiler algorithms for synchronization strength reduction is appropriate. However, we also believe that having point-to-point synchronization available explicitly within the languages is important to avoid performance loss when compiler optimization is inadequate. [...]

The SL constructs for synchronization using explicit dataflow channels, and our choice to expose the b, p and a operations for consistency, go along with this suggestion.

7.8.4 ZPL's Ironman interface

ZPL [Cha01] was a precursor to Chapel [CCZ07] designed at the University of Washington. The designers of ZPL were already highly concerned by issues of memory consistency over weakly coherent systems and distributed memories. Like us, they were interested to both support distributed memory and automate the elision of communication primitives when communication partners were running in the same memory domain. Their solution was the Ironman interface, introduced in [CCS98] and detailed in [Cha01, Chap. 4]. As we do in section 7.3, this interface proposes abstract primitives for use in the code generated by compilers; the primitives are then projected to different implementations in the run-time environment depending on the underlying communication infrastructure. The primitives in Ironman are, on the writer side:

SR (Source Ready) : The values at the source processor will not be written again prior to transfer. The source processor is now ready to begin the data transfer.

SV (Source Volatile) : The data at the source processor is about to be overwritten. Execution cannot continue until the transfer is completed.

Our proposed p operation combines the function of Ironman's SR and SV operations. The primitives on the reader side are:

DR (Destination ready) : The locations at the data destination will not be used again until the transfer has completed. The destination processor is now ready to accept data from the source processor.

DN (Destination Needed) : The values at the data destination are about to be read. Execution cannot continue until the data from the source processor has been received.

Our proposed *a* operation combines the function of DR and DN.

The Ironman primitives are designed to facilitate the overlap of communication with computation, by separating issuing a communication operation from completing it. As such they are similar to e.g. MPI's asynchronous send/receive APIs; their difference with an API is that they can be elided at run-time by direct shared memory synchronization if the communication endpoints are close to each other.

These primitives further assume that any address in the local address space can serve as communication buffer. In particular, they does not offer a handle on system that require buffers to be allocated in a dedicated memory space, like our *b* and *a* operations do.

Summary

Abstract machines establish a contract between a platform provider and the designers of higher-level abstractions, including programmers. This should be provided and detailed, so as to create an abstraction interface where external observers can reason about program semantics independently from specific implementations.

□ In our work, we have designed such an abstract machine to describe the semantics of the platform introduced in part I and chapter 5. The abstract machine describes dataflow synchronization and establishes memory consistency as a derived property of the directed acyclic graph of the scheduling order in programs. It also introduces "communication and consistency domains" to expose different kinds of implicit communication using hierarchical

▷ shared memory systems. We have postulated that the new concurrency semantics added to the latest revisions of the C and C^{++} standards can be emulated using the proposed abstract machine.

Chapter 8

Configuration of the visible synchronizer window

—Highlight on specifics, opus 1

Abstract

Some architectural innovations *may be eventually invisible to application developers* and thus not part of the "advertised" main features of a design. Yet they may significantly impact issues under control of operating software providers, and thus must be fully described for this audience. The proposed machine interface from chapter 4 illustrates this. In traditional register machines, the number of register names available for use in programs is fixed by the ISA definition. In our case, the code generator that produces the machine code can *choose* the number of register names available for use: as per sections 3.3.3 and 4.3.3, the hardware dynamically maps register names to physical synchronizers. The set of mapped synchronizers constitutes a "visible window" from the perspective of individual threads. A lower requirement on the number of available register names enables more compact use of the synchronizing storage, potentially more thread contexts active, and thus potentially better opportunities for latency tolerance. In this chapter, we describe to operating software providers how a code generator can configure the layout of the visible window.

Contents

8.1 Introduction

The machine interface in chapter 4 allows software to configure a separate register window layout for each thread program. Specifically, a code generator must produce a triplet of values G, S, L in the machine code for each thread program. These specify the number of synchronizers that must be made available by the architecture at run-time before execution of the thread program starts; respectively, the number of "global" synchronizers visible from all thread contexts, the number of "shared" synchronizers shared by adjacent contexts, and the number of "local" synchronizers private to each context (cf. figs. C.1 and C.2 for an example). These values are constrained only by the equation $G + 2S + L \leq M$, where M is the total number of register names available for use by programs in the ISA, commonly 31. The question thus arises of how to determine these values.

We have explored multiple strategies. In this process, we determined the general upper and lower bounds on these values, as well as the design spectrum of possible solutions, which we describe. Our implementation uses the outcome of our findings, described below. We distinguish in our text further between "synchronizers," which are the physical units of storage, and "register names," which are the addresses used as operands in machine instructions.

8.2 Formalism

□ We start by formalizing as follows:

- we define the set $\mathcal{P}$ of all machine representations for thread programs on the architecture, and
- the set $\mathcal{F}$ of all machine representations for regular functions (which can be *called* from a thread, not executed as a family of threads).
- Then for any $P \in \mathcal{P}$ we can define:
  - $L(P)$, the number of register names reserved as local synchronizers in the declared interface (cf. section 4.3.3);
  - $G(P)$, the number of register names used for incoming "global" channel endpoints (cf. section 4.3.3.2);
  - $S(P)$, the number of register names used for outgoing "shared" channel endpoints (cf. section 4.3.3.3);
  - $R(P)$, the highest register name actually used as local synchronizer in P ($R \leq L$);
  - $D_c(P) \subset \mathcal{P}$, the set of thread programs that are used as the target of family creation in P;
  - $D_f(P) \subset \mathcal{F}$, the set of functions called directly from P;
- and for any $F \in \mathcal{F}$ we can define:
  - $R(F)$, the highest register name actually used as local synchronizer in F;
  - $D_c(F) \subset \mathcal{P}$, the set of thread programs that are used as the target of family creation in F;
  - $D_f(F) \subset \mathcal{F}$, the set of functions called directly from F.

- We also define $D_f^*(x) \subset \mathcal{F}$, the set of all functions called transitively from x (from either $\mathcal{P}$ or $\mathcal{F}$), defined by:

$$\begin{aligned} D_f^0(x) &= D_f(x) \\ D_f^{n+1}(x) &= \{D_f(y) \| y \in D_f^n(x)\} \\ D_f^*(x) &= \bigcup_{n\to\infty} D_f^n(x) \end{aligned}$$

  (This set is finite on any actual implementation of the architecture due to the finiteness of the address space)

8.3 General constraints

□ From the definitions above, we are able to formulate the following constraints on G, S and L:

1. *outer interface fit*:
$$\forall P \in \mathcal{P} \quad G(P) + 2S(P) + L(P) \leq M$$
This denotes that the window specification must fit the maximum size in the substrate ISA. The factor 2 for S comes from the fact that there are two sets of "shared" synchronizers: those shared with the previous context and those shared with the next context (cf. figs. C.1 and C.2).
2. *required lower bound for calls*:
$$\forall F \in \mathcal{F}$$
$$\left(\max_{F' \in D_f^*(F)} R(F')\right) \leq R(F)$$
$$\forall P \in \mathcal{P}$$
$$\left(\max_{F \in D_f^*(P)} R(F)\right) \leq R(P) \leq L(P)$$
This denotes that must be enough private synchronizers to allow the code of any directly and indirectly called functions to run within the same context.
3. *required lower bound for creations*, in interfaces with "hanging" synchronizers (section 4.3.3.3):
$$\forall P \in \mathcal{P}$$
$$\left(\max_{P' \in D_c(P)} (G(P') + S(P') + 1)\right) \leq R(P) \leq L(P)$$
$$\forall F \in \mathcal{F}$$
$$\left(\max_{P' \in D_c(F)} (G(P') + S(P') + 1)\right) \leq R(F)$$
This denotes that there must be enough private synchronizers to serve as channel endpoints for all the created families.

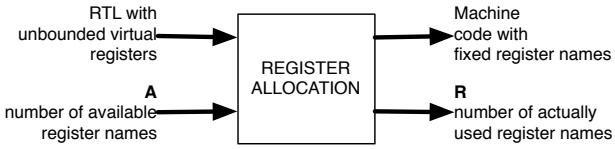


Figure 8.1: Register allocation as a function of the number of available register names.

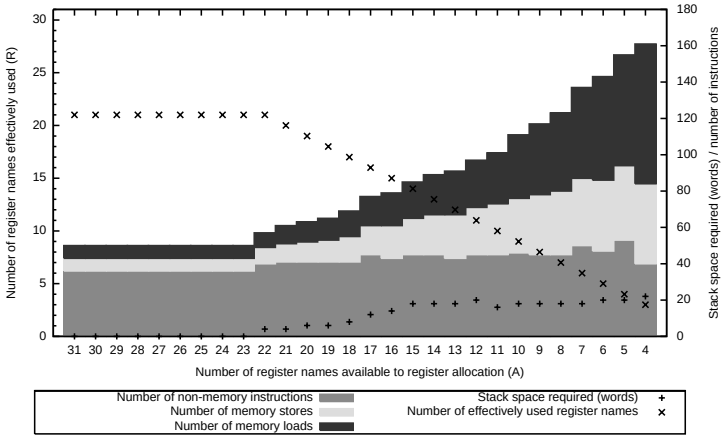


Figure 8.2: Effect of various numbers of available register names on the outcome of register allocation with a simple function updating a variable in memory.

4. *required lower bound for creations*, in interfaces with "separated" synchronizers (section 4.3.3.3):

$$\forall P \in \mathcal{P} \quad \begin{cases} 0 \le R(P) \le L(P) & \text{if } D_c(P) = \varnothing \\ 1 \le R(P) \le L(P) & \text{otherwise.} \end{cases}$$

$$\forall F \in \mathcal{F} \quad \begin{cases} 0 \le R(F) & \text{if } D_c(F) = \varnothing \\ 1 \le R(F) & \text{otherwise.} \end{cases}$$

8.4 Degree of freedom from register allocation

We focus here on the distinction between the number of *available* register names, which is the value declared with L, and the number of *effectively used* register names R in a given machine representation.

To understand their relationship, we must consider how register allocation works in a compiler. To summarize, a compiler considers the maximum number of simultaneously active local (C's "automatic") variables in a function body, together with intermediate results in expressions, and deduces the required number of *frame slots* N for a function or thread program body. N is independent of the target architecture configuration. Then the variables that are most used are mapped onto A available register names, where A is typically configurable. We illustrate the functional nature of register allocation in fig. 8.1.

If $N \leq A$, then $R = N \leq A$ register names are used. If $N \geq A$, then $R = A$ register names are used and the code generator inserts extra instructions to save and restore value slots to and from memory. We illustrate this in fig. 8.2, where a single C function performing arithmetic on an array is processed through various values of A. Traditionally, A is fixed by subtracting a number of "reserved" names (e.g. stack pointer) from M. It then becomes an architectural constant valid for all code generators on that architecture. For our architecture, we can choose both L and A, with the constraint that $A \leq L \leq M - G - 2S$.

In short, next to G, S and L there is an additional degree of freedom A which is an input to code generation.

8.5 Strategies for synchronizer mapping

□ We considered only strategies for synchronizer mapping that account for separate compilation, that is, a given thread program or C function may be compiled without knowing the definition of all other thread programs or functions that it composes through family creation or C function calls. We acknowledge that there exist other strategies when all the program source is visible during code generation, but we decided early on to focus first on providing a platform where separate compilation is possible.

With separate compilation, any strategy to compute a synchronizer mapping for a function needs to account for the possibility that the resulting machine code may be composed in a yet unknown program context. This requires any strategy to select G and S using only the prototype declaration of the thread program being considered, since this is the only information available from another translation unit during separate compilation. In particular, *it is not possible to choose G and S as a function of L*, since L will not be known in another translation unit while G and S will be required to generate code for family creations.

Furthermore, to ensure compilation *soundness*, the strategy should not cause a valid program composition in the semantics of the source language to have no valid representation in the machine code.

We considered first the numbers G_s and S_s of "global" and "shared" channel endpoints defined in the *program source*. Then we considered the simplest possible strategy, i.e. use $G = G_s$, $S = S_s$, then choose $A = L = M - G - 2S$. There are three problems with this strategy. To understand why, we first distinguish *fitting* programs where $G_s + 2S_s \leq M$ and *overflowing* programs where $G_s + 2S_s > M$. Then:

1. The strategy is "synchronizer-greedy," since a simple thread program which may only use a few synchronizers and channel endpoints will still end up reserving all M register names from the architecture.
2. The strategy is not sound. There are two situations:
   - with "hanging" synchronizer mappings (section 4.3.3.3), once a fitting program P_1 is compiled, it becomes impossible to separately compile another program P_2 which creates a family running P_1 if $G(P_1)+S(P_1)+1 \geq M-G(P_2)-2S(P_2)$, even though P_2 on its own may be fitting. In other words, this strategy establishes that *whether* a thread program can be used to create a family is a function of the interface of the creating thread.
   - with "separated" synchronizer mappings, once a fitting C function F is compiled ($R(F) \leq M$), it may be impossible to separately compile a thread program P which calls F if $R(F) \geq M - G(P) - 2S(P)$, even though P on its own may

be fitting. (The difference with the previous argument is that it is now plain C function calls, instead of thread creations, that defeat soundness.)

3. The strategy requires the code generator to reject overflowing programs. This is unsatisfactory, because it requires the programmer to know M to decide whether a program is valid, and it establishes a lower bound on M for future architectural changes as soon as a program is written.

We addressed these problems in turn, as described below, to establish the strategy we finally used in our implementation.

8.5.1 Minimizing the number of local synchronizers

□ To solve the first problem, we can choose $A = M - G - 2S$ first, let register allocation run and use the resulting R as a value for L (this is possible since $R \leq A$). The alternative is choosing $A < M - G - 2S$, to reduce the mandatory synchronizer cost of the thread program even further. However, as seen in the previous section, when A becomes smaller, the code generator must introduce extra memory load and store instructions for spills. This means that using A to control the synchronizer cost is effectively a trade-off between this cost and memory access penalties. To distinguish the benefits and drawbacks, we must therefore look more closely at the hardware implementation:

- in an *interleaved* multithreaded execution (one pipeline, one synchronizer store), adding memory operations is always a penalty, even if all memory latencies can overlap with useful computations, because the total number of executed instructions increases. In this situation, the largest value $A = M - G - 2S$ should be used to minimize the number of memory accesses in all cases.
- in a *simultaneous* multithreaded execution (multiple pipelines sharing one synchronizer store), A should be reduced so that there can be enough threads to keep all functional units busy and also tolerate the latencies of the added memory latencies.

Since the implementations we used have one in-order pipeline per synchronizer store, we choose $A = M - G - 2S$ accordingly.

8.5.2 Ensuring sound composition

8.5.2.1 Case with "hanging" synchronizer mappings

□ We consider first "hanging" synchronizer mappings as introduced in section 4.3.3.3. In this case, any upper bound for G and S for a given program P becomes a lower bound on the choice of L for any other program that wishes to create a family running P. A lower bound on L is in turn an upper bound on G and S in that other thread. Since any solution must ensure compatible values of G, S and L for family compositions across separate translation units, we must establish a *fixed point* for the upper bound on G and S, and the lower bound on L. Let us call this fixed point X. By definition, $\forall P \in \mathcal{P} \quad G(P) + 2S(P) \leq X$ and $G(P) + S(P) + 1 \leq L(P) \leq M - X$.

Furthermore, X must be independent from specific program sources, since X must be known by convention across all separate compilations. This means that X can be chosen arbitrarily. Once the fixed point is chosen, we can *deduce from* X the upper bounds on G, S and L as per the definition above.

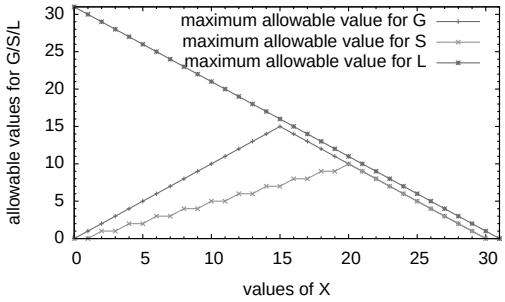


Figure 8.3: Maximum allowable values of G, S, L for various values of X, with $M = 31$.

Configurations suitable for "hanging" synchronizer mappings, as per section 4.3.3.3.

Since register allocation benefits from large values of L, and inter-thread communication benefits from large values of G and S (more asynchrony), we are thus interested to choose X in a way that maximizes L, G and S. To clarify, we plotted in fig. 8.3 the upper bounds for all values of X with $M = 31$. As can be seen in this diagram, all values of X below 15 enable more communication endpoints at the expense of local synchronizers. Values between 15 and 20 allow more "shared" channel endpoints at the expense of both "global" channel endpoints and local synchronizers. Values above 20 restrict all three values further.

The optimization of X would require empirical results across a set of applications. Indeed, a low value of X can require inter-thread communication to occur more often via memory instead of the architecture's dedicated channels, while a high value of X can require more spills to memory due to a smaller number of private synchronizers. The best trade-off for a given application depends on the source code of the entire application and how program components are composed together; the best trade-off for an entire application domain requires empirical evaluation across the entire domain.

Without considering a specific application domain, we made X configurable and set it by default to $X = 12$. We motivated this default value as follows. First we observed that a prospective user of our technology, the SAC2C compiler for Single-Assignment C [GS06], was structured to make heavy use of common loop variables, which would be translated to "global" channel declarations in our interface language. This suggested maximizing the upper bound on G, and thus choosing the largest possible value of X while keeping $X \leq 15$. Then we observed that the calling conventions for procedure calls on existing general-purpose processors with RISC ISAs often choose to use 4 to 8 register names to argument passing (table 8.1), plus 6 to 12 "scratch" caller-save register names for temporary variables. Then we deduced from this observation that the designers of these calling conventions must have empirical knowledge that this is an optimal trade-off to support C functions and calls in general-purpose workloads. So we adopted this requirement in our setting, and we deduced the constraint $L \geq 18$, hence $X = 12$.

Once X is set, the resulting code generation strategy can be summarized as follows: first set G and S from G_s and S_s while ensuring $G + 2S \leq X$; then run code generation with $A = X$, and finally set $L = R$ after code generation. This is the implementation we describe in Appendix H. The procedure calling convention within threads is left unchanged from the substrate ISA.

ISA	ABI	Integer arg. regs.	FP arg. regs.
PowerPC	(all)	3-10	33-40
Alpha	(all)	6	6
SPARC32	System V, Sun	6	0
SPARC64	Sun	6	16
MIPS32/64	"old"	4	2
MIPS32/64	"new"	8	8
ARM	Standard [ARM09]	4	8
x86-64	Unix	6	8
x86-64	Microsoft	4	4

Table 8.1: Calling conventions on existing general-purpose processors.

Information extracted from the GNU CC target machine configurations.

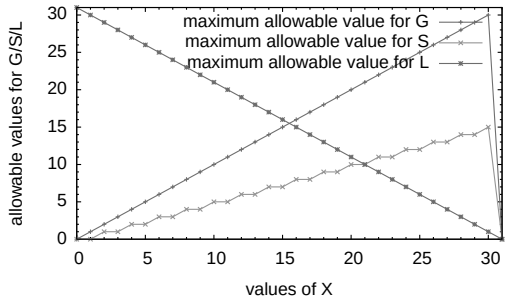


Figure 8.4: Maximum allowable values of G, S, L for various values of X, with $M = 31$.

Configurations suitable for "separated" synchronizer mappings, as per section 4.3.3.3.

8.5.2.2 Case with "separated" synchronizers

□ We consider here "separated" synchronizer mappings as per section 4.3.3.3. Any upper bound for A (or R) for a given C function F becomes a lower bound on L for a program P that wishes to call F. A lower bound on L is in turn an upper bound on G and S in P. Since any solution must ensure compatible values on G, S, L and R for function calls across separate translation units, we must establish a fixed point for the upper bounds on G and S, and the lower bound on L. Let us call this fixed point X. By definition, $\forall P \in \mathcal{P} \quad G(P)+2S(P) \leq X$ and $1 \leq L(P) \leq M - X$.

As described before, X must be known by convention, and once known it can be used to derive L, G and S in thread programs, and A (or R) in C functions. As before, we are interested to choose X in a way that maximizes L, G, S, and R for C functions. With the new constraint, we plotted in fig. 8.4 the upper bound for all values of X with $M = 31$. As can be seen in this diagram, increasing X allows for more channel endpoints as the expense of local registers.

Again, the optimization of X requires empirical data across a target application domain. The trade-offs remain unchanged, as well as the methodology. We thus felt justified to reuse the same semi-arbitrary choice of $X = 12$ in our setting, for the same reason as above. This allows us to keep the previous decision strategy for G, S and L unchanged.

```
1  sl_def(foo, sl_glparm(T1, N1), ... sl_glparm(Tn, Nn)) {
2    ... sl_getp(N1) ... sl_getp(Nn) ...
3  };
4  ...
5  {
6    sl_create(..., foo, sl_glarg(T1, N1, Vn),
7                  ... sl_glarg(Tn, Nn, Vn));
8    sl_sync();
9  }
```

Listing 8.1: Code fragment using multiple virtual "global" channels.

```
1  struct foo_i { T1 N1; ... Tn Nn; };
2  sl_def(foo, sl_glparm(struct foo_i *, __mp)) {
3    ... sl_getp(__mp)->N1 ... sl_getp(__mp)->Nn ...
4  };
5  ...
6  {
7    struct foo_i __mp = { V1, ... Vn };
8    sl_create(..., foo, sl_glarg(struct foo_i*, , &__mp));
9    sl_sync();
10 }
```

Listing 8.2: Translation of listing 8.1 to use only one "global" channel.

Note that we use here the constraint between X, G and S stemming from program composition via C function calls under separate compilation, whereas the constraint we used in section 8.5.2.1 stemmed from composition via thread creations. The reason why we did not use the same constraint in both sections is that the constraint from composition via thread creation is more restricting with "hanging" synchronizer mappings, whereas it is less restricting with "separated" mappings.

8.5.3 Handling overflowing programs

□ The last problem was dealing with programs where the number of expressed channel endpoints is too large for the code generation constraints. In our case, with $X = 12$, we can use at most 12 hardware endpoints for "global" channels directly, or 6 endpoint pairs for "shared" channels, or any combination where $G + 2S \leq 12$. We could propagate these constants to the input language definition, and thus make programmers (or code generators) aware of this hardware limit and accept the invalidity of overflowing programs as a design choice; however, an additional improvement is possible.

Indeed, we can reuse the strategies from [JR81] to support *virtual* inter-thread channels on the new architecture. With only one physical "global" channel endpoint, it is possible to emulate a virtually unbounded number of other "global" channel endpoints by storing the source values in memory instead. For example, the program fragment in listing 8.1 can be replaced by listing 8.2 without any changes in semantics. In general, any number of "global" endpoints declared in source can be mapped to $G \geq 1$ endpoints in hardware, where the first $G - 1$ hardware endpoints are mapped directly to program-specified endpoints, and the last

endpoint in hardware is mapped to a pointer to a data structure in memory that contains the source values for the other program-specified endpoints. We call this method *escaping* the channel source values into memory, because it trades physical hardware channels for memory capacity. When we use escaping, no value of G_s causes the program to be overflowing as soon as $G \geq 1$ is allowed by X. We actually implemented this method in our solution.

The method can be further extended to "shared" channels by reserving a data structure in memory in each logical thread in a family to hold the source values for the next logical thread.

However, here a difficulty arises. We cannot translate multiple individual uses of `sl_setp` that manipulate different virtual "shared" channel names to multiple uses of `sl_setp` that manipulate the same hardware channel endpoint. This is because the machine interface does not allow us to reset the synchronization state of the channel, i.e. we must assume the channel only synchronizes once. This requires us to replace all escaped uses of `sl_setp` by non-synchronizing memory assignments, and *also* add a single use of `sl_setp` at the earliest point of the control flow graph that is guaranteed to run after all the memory assignments.

▷ Unfortunately, this would be a program transformation that requires semantic analysis of the program source, which is outside of the scope of the framework we discussed in chapter 6. For this reason, we originally chose to not extend the method to "shared" channels in our implementation. Then we realized that there exists an alternative, to add the trailing `sl_setp` at the end of the control flow graph, i.e. at the end of the function body and at every use of `return`. This would be a suitable context-free substitution. The drawback of this method is that it over-synchronizes by forcing the sequentialization of all participating threads. If proper virtualization of "shared" channels is desired, this technique could be added to our implementation at a low cost. The reason why we did not yet implement this solution is suggested in section 13.8.

Our resulting implementation can be summarized as follows:

- in the language specification (Appendix I, clause I.4.2§2), we guarantee support for 127[1] "global" channel endpoints and 1 "shared" endpoint pair;
- in the actual implementation, the number of "global" endpoints is effectively bounded only by the memory available to the creating thread, and the number of "shared" endpoint pairs is indirectly bounded by X, to 6 pairs in our case; however, we do not advertise this value so as to enable other implementation-dependent choices for X.

Summary

Variably sized visible windows influence register allocation during code generation. To enable separate compilation, a static limit must be set on the number of register names available to register allocation in code generators for local variables, which in turn constrains the number of register names available to define synchronization channel endpoints.

□ ▷ In this chapter, we have formalized the possible trade-offs in this choice. We then described how to virtualize an arbitrary number of logical channels defined in programs over a fixed set of register names in the ISA. The result is a configuration strategy that is mostly invisible from the perspective of the C programmer or higher-level code generator.

[1] For symmetry with the limits on the number of function arguments in [II99, 5.2.4.1§1] and [II11b, 5.2.4.1§1].

Chapter 9

Thread-local storage
—Highlight on specifics, opus 2

Abstract

Some functional requirements of operating software providers may be both *invisible to hardware architects* and *invisible to application developers*; they may exist to support the *generality* of the operating software or its *reusability* across platforms. From the hardware architect's perspective, an opportunity exists to learn about these requirements and optimize a design accordingly. For example, Thread-Local Storage (TLS) is a necessary feature of run-time environments when threads must run general-purpose workloads. This is fully under the responsibility of the operating software, as TLS is "just memory" from the hardware provider's perspective, and assumed to pre-exist by application developers. Meanwhile, on a massively concurrent architecture which encourages the creation of many short logical threads in hardware, such as the one introduced in part I, provisioning TLS via traditional heap allocators would be largely detrimental to performance and efficiency due to contention and over-allocation. In this chapter, we show the gains to be obtained by co-designing support for TLS between the hardware architect and the operating software provider. We provide an analysis of the requirements to provision TLS on such architectures and a review of the design space for solutions. We illustrate with our own implementation.

Contents

9.1 Introduction

A C compiler for most register-based machines uses a stack pointer for two purposes:

- to allocate and access function activation records[1].
- when using the commonly-supported C extension "`alloca`" or [II99]/[II11b]'s variably sized arrays scoped to a function body, to perform the corresponding dynamic allocation at run-time.

In addition to this, most existing C compilers support the notion of "thread-local variables," declared with a special storage qualifier ("`thread`" or similar), and which are guaranteed to designate, at run-time, a different object in each thread whose initial value in each thread is the initial value defined in the program at the point of definition. This definition has been captured in the latest specifications [II11a, II11b]. The implementation of this feature is based on a "thread local storage template" generated by the compiler statically, and passed by reference to each newly created thread to be duplicated in a thread-local memory.

Activation records, dynamic local allocation and thread-local variables define collectively the notion of "Thread-Local Storage (TLS)," which is an area of memory that can be used by thread program code with the assumption that no other thread will inadvertently overwrite its values. In general terms, TLS is required for a thread to properly virtualize a general-purpose computer with the computing abilities of a Turing machine (cf. section 1.2.1). Conversely, an execution environment must provision TLS to claim its support for general-purpose workloads.

In this chapter, we examine how TLS can be provisioned to threads efficiently in an architecture with fine-grained concurrency, and how access to TLS can be gained from the perspective of code generation.

9.2 "Traditional" provisioning of TLS

The two traditional strategies to obtain access to TLS in new threads are: *dynamic pre-allocation*, before the thread starts by the creating thread, or *self-allocation* by the newly created thread after it has started. With pre-allocation, a pointer to TLS can be passed as an argument, or at a predefined location. With self-allocation, a protocol must exist so that a thread can *request* the allocation of TLS.

9.2.1 Dynamic pre-allocation

Dynamic pre-allocation requires *a priori* knowledge of an upper bound on the TLS size. We found an extensive analysis of this in [Gru94]. To summarize, a combination of compiler-based analysis of the call graph in each thread program, together with a profile-directed analysis of the actual run-time requirements, can provide a conservative estimate of this upper bound in a large number of applications.

There are two issues with pre-allocation however. The first issue arises when a primitive exists to bulk create many logical threads over a limited set of hardware threads, such as presented in part I. Each logical thread potentially needs TLS. This has two consequences:

[1] Activation records primarily contain local variables that do not fit in registers, e.g. local arrays, and also possibly spill space and the return address in called procedures.

- since the mapping of logical threads to hardware thread contexts may not be visible in software, potentially as many separate TLS spaces would need to be allocated as there are logical threads in the family. This would cause severe over-allocation for large logical families running over a few execution units;
- any use of pre-allocation would require two values to be passed as argument (a base pointer and a size) and would mandate either a potentially expensive multiplication with the logical thread index in every thread, or sizes that are powers of two, to allow a cheaper shift operation instead.

The second issue is that pre-allocation is not an appropriate general scheme: it is unable to provide TLS for recursion where the depth cannot be statically bound, for example data-dependent recursions, and it cannot cater to dynamic allocation of local objects whose size is known only at run-time. Because of this latter issue, whether pre-allocation is used or not, a fully general system would thus need to also offer self-allocation as an option.

9.2.2 Self-allocation

Self-allocation, in contrast, seems relatively simple to implement. It suffices to emit, at the start of every thread program that requires TLS, an invocation of a system service in charge of dynamic memory allocation. Supposing such a service exists, we explore here how its interface should look. There are two possible strategies. Either the service is implemented by means of a regular procedure, which is called by branching control within the same thread, or it is implemented as a thread program, which is accessed by creating a new thread running that program. In both cases, some form of mutual exclusion over a shared memory allocator is required; otherwise, the two strategies differ as follows.

The procedure-based approach requires an allocation service that can be run from individual threads at a low cost. To keep logical thread creation lightweight, requirements such as a privilege switch or a trap to access the service would be inappropriate. The allocator would be a procedure that negotiates exclusive access to some data structure, determines a range of addresses to use as TLS, and releases the data structure. For best performance this should have a fixed, low overhead in most cases. In turn this requires minimizing non-local communication, hence requires a pooled memory allocator with per-processor pools. This is the approach taken by e.g. the "polo stack" allocation strategy from [Abd08, Chap. 6].

The thread-based approach creates the opportunity to delegate the allocation to some other resource on the system. This is beneficial as the designer may not wish to equip every processor with the hardware facilities required for a memory allocator, yet desire to allow every logical thread to access TLS. This is of course constrained by contention: the system must provision enough bandwidth to this service, either through faster processors implementing the service or a larger number thereof.

Then, the service interface must guarantee that the thread creation is always possible[2]. This in turn mandates that there is at least one thread context which is never automatically used by bulk creation except for TLS allocation requests, and that the program must be able to force the hardware "allocate" request to grab that specific context. This in turn requires a mechanism in hardware to reserve a context permanently and an addressing scheme to identify that context in the "allocate" operation.

[2]The scheme described later in chapter 10 does not apply here as it pre-requires TLS to serialize the behavior locally.

To summarize, TLS self-allocation is a viable strategy which can either be implemented locally on each processor, or on service processors shared between multiple multithreaded cores. It requires mutual exclusion and an allocation algorithm with an extremely low overhead, so that thread creation stays cheap. Contention may be an issue when sharing the service between multiple cores, a topic that we revisit later in section 14.4.

9.3 Smarter provisioning of TLS

Dynamic pre-allocation and self-allocation require the time overhead of a procedure call or one extra thread creation and synchronization during the start-up of every logical thread. This overhead, commonly between 50 and 200 processor cycles on contemporary hardware (for procedure calls) or 50 cycles on the proposed architecture, does not compare favorably with the low logical thread activation overhead on the reference implementations of our new architecture (less than 5 processor cycles per logical thread). Moreover, since the machine interface does not guarantee that the values of the local synchronizers persist across logical threads sharing the same thread context, it would be necessary to either design a new architectural mechanism to carry the TLS pointers from one logical thread to the next, or invoke the allocation service in every logical thread, not only the first. We present two alternative strategies, presented below.

9.3.1 Persistent dynamic allocation

□ The first strategy we considered preserves a TLS range across logical threads sharing the same thread context. In this approach, each thread context retains, next to the program counter, a base pointer for the area of TLS reserved for all logical threads sharing the context. This is a valid approach since all logical threads execute in sequence without interleaving (sections 3.3.1 and 4.2). When the first logical thread in a family starts in a context, it checks this base pointer, and performs an initial allocation if the base pointer is not yet configured. All subsequent logical threads in the same context can reuse the same TLS[3].

Here there are two storage reclamation strategies: *aggressive* reclamation, where TLS is de-allocated as soon as the context is released, and *delayed* reclamation where TLS remains allocated between families.

To enable aggressive reclamation, either a hardware process, or the thread synchronizing on termination of a family, would explicitly de-allocate all the TLS areas allocated for each thread context effectively used by the family. This has linear time complexity with the number thread contexts used for the family per core, and is independent of the logical thread sequence which is typically larger. The cost of allocation is then repeated for each new created family.

With delayed reclamation, TLS space is reused across families: the cost of allocation is factored at the expense of possible over-allocation. There is a spectrum of implementation strategies. Asynchronous Garbage Collection (GC) of TLS areas for inactive thread contexts is possible if the application code guarantees no sharing of TLS data between threads. Otherwise, synchronous GC can be used, or the operating system can reclaim storage when the entire program terminates.

[3]We assume here that all threads have the same TLS size requirements. Otherwise, the size should be stored alongside the base pointer, compared upon thread startup, and the TLS should be reallocated when a new size is desired.

9.3.2 Context-based pre-reservation

□ We pushed the concept of persistent allocation with delayed reclamation further: we explored more static schemes where TLS space is pre-allocated for all thread contexts on a group of processors before a software component is assigned to this group.

To optimize this situation, we propose to allow a logical thread to construct a pointer to a region of TLS from the identity of the thread context where the logical thread runs. We experimented with multiple implementations, with various trade-offs between the latency benefit and the cost of the required extra logic. We determined two orthogonal design decisions: how the space is *allocated* from shared storage, and how the addresses are *distributed* to threads.

Regarding allocation, we found two implementable strategies:

- *static* pre-allocation: a predetermined amount of storage is allocated for each thread context. This is suitable if pre-allocation is possible, i.e. when there are known upper bounds on TLS size and the provision of TLS for all contexts fits in the available storage;
- *deferred* pre-reservation using virtual addressing: a predetermined number of virtual addresses are reserved for each thread context, then storage is allocated on-demand when the addresses in the TLS areas are first used. Although this scheme also bounds the amount of TLS via the number of addresses reserved, it is possible to make this bound so high that the addresses available to each thread context would be sufficient to address the entire physical storage.

Regarding distribution, we found two implementable strategies:

- *external* distribution: a table exists in the system, indexed by the logical processor identifier and thread context identifier of each thread, which can be configured to contain a different address and size for TLS for every thread context. This table can reside in main memory with a base address determined by convention, or in dedicated memory structures on chip next to each processor;
- *computed* distribution: a common base pointer and TLS size are accessible to multiple thread contexts by convention, and can be configured in software. In each thread, the program can compute a private area of TLS by multiplying a combination of its logical processor identifier and thread context identifier with the common size, and adding this to the common base pointer. The pair (base pointer, size) can be shared by all processors in the system, or configurable by processor.

We summarize the trade-offs of these four combinations in table 9.1.

The external/static and external/deferred schemes are the schemes used in most existing general-purpose operating systems to implement processes and threading. For example in GNU/Linux, Solaris and BSD each system thread is allocated a range of virtual addresses as stack, which is populated on demand while the thread runs. In uCLinux for embedded systems without virtual addressing, processes receive a dedicated subset of the physical memory as TLS.

9.4 Implementation

We did not implement dynamic pre-allocation in our technology (cf. chapter 6). We also did not thoroughly explore self-allocation or any scheme requiring mutual exclusion, including

		Distribution	
		External	Computed
Allocation	Static	Thread contexts compete for a subset of the physically available memory, but heterogeneous sizes are possible. Time-expensive to pre-configure (linear in the number of contexts), cheap in logic (needs access only to the context identifier).	All thread contexts compete equally for a subset of the physically available memory. Cheapest in time and extra logic to pre-configure (constant time, needs access only to the context identifier), least flexible for software.
	Deferred	Thread contexts compete for a subset of the virtual address space, and heterogeneous address windows are possible. Most time-expensive and logic-expensive to pre-configure (linear in the number of contexts, requires virtual addressing and handling of translation faults), most flexible for software.	All thread contexts compete equally for a subset of the virtual address space. Cheap in time to pre-configure (constant time), expensive in extra logic (requires virtual addressing).

Table 9.1: Trade-offs of context-based TLS partitioning

Side note 9.1: Support for heap-based dynamic allocation.

Although we did not study self-allocation for obtaining TLS space in threads, we did not deliberately avoid dynamic memory allocation altogether. Indeed, our software integration work eventually included a two stage allocator into the standard C library available to programs, to support the "`malloc`" API. This allocator handles small sizes locally using binned allocation of fixed sizes chunks in TLS-based "local heaps," and delegates allocation of larger sizes centrally by an off-the-shelf allocator using mutual exclusion, Doug Lea's `dlmalloc` [Lea96]. Our allocator is available publicly with our SL software.

persistent dynamic allocation from section 9.3.1. We made this decision for two reasons. First, as highlighted at the start of section 9.3, the overhead of dynamic allocation is high compared to the latency of thread creation, and we were interested in exploring strategies that provide TLS at lower latencies. Then, the system implementation we had at our disposal did not provide facilities for mutual exclusion until a much later phase, where our techniques from section 9.3 were already implemented and demonstrated to perform adequately.

Considering the context-based pre-reservation schemes from section 9.3.2, we also avoided the schemes based on external distribution. We made this choice because we were focusing on parallel execution over large numbers of processors, and we predicted that external distribution would incur large pre-configuration costs every time a group of processors is recycled between benchmark applications.

Instead, during our work we explored the two remaining context-based schemes that we have identified: static/computed and deferred/computed. We found them attractive because they are simple to implement and the computed access to TLS makes the preconfiguration costs scalable to large numbers of processors and thread contexts.

Physical RAM reserved for TLS	Number of processors	Contexts per processor	TLS size
256 KiB	1	64	4 KiB
16 MiB	1	64	256 KiB
256 KiB	8	8	4 KiB
16 MiB	8	8	256 KiB
1 GiB	8	8	16 MiB
256 KiB	64	64	64 B
16 MiB	64	64	4 KiB
1 GiB	64	64	256 KiB

Table 9.2: TLS size per thread with the static/computed scheme.

9.4.1 Static/computed partitioning

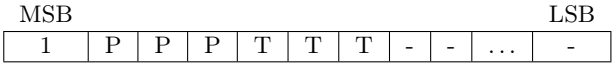
The first scheme we implemented was a context-based, static computed partitioning of the address space. In this scheme, the operating system allocates a fixed subset of the physical RAM and divides it equally across all thread contexts. This scheme does not allow the TLS space of a thread context to be expanded, with the same drawbacks as presented in section 9.2.1.

We present possible configurations in table 9.2. As can be seen in this table, the static/computed strategy does not scale well for larger numbers of processors and/or thread contexts per processor. It incurs significant wasted storage space, caused by different TLS requirements between logical threads: the common size must be large enough to cater to the highest requirement, which is typically uncommon at run-time. The amount of wasted space increases with the number of processors and/or the number of contexts per processors.

Nevertheless, this was the scheme eventually used to test the UTLEON3-based platform [DKK+12], as this implementation contains only 1 core.

9.4.2 Deferred/computed partitioning

The second scheme we implemented was context-based, deferred/computed partitioning of the address space with virtual addressing. We made the assumption that the entire system shares a single virtual address space, i.e. that the same address translation unit is used by all processors and the inter-processor memory network uses virtual addresses. Then we used the following address format for TLS:



In this scheme, a thread can construct a valid address for its TLS area by setting the Most Significant Bit (MSB), then concatenating its logical processor index and thread context index in the most significant bits of the address. Since this information is already present in each processor's pipeline to support scheduling, this value can always be constructed cheaply (one or two pipeline cycles). The lower half of the address space (with the MSB unset) stays available for common data, code, shared heaps, system data structures, etc.

We document the maximum allowable TLS size per thread under various architecture configurations in table 9.3. Despite the static address space reservation, with 64-bit addressing there is sufficient address space capacity in each thread to potentially address the

Number of address bits	Number of processors	Contexts per processor	Maximum TLS size
32	8	8	32 MiB
32	64	64	512 KiB
32	1024	256	8 KiB
64	64	64	2 EiB
64	1024	256	32 PiB
64	1024	1024	8 PiB

Table 9.3: Maximum possible TLS sizes with the deferred/computed scheme.

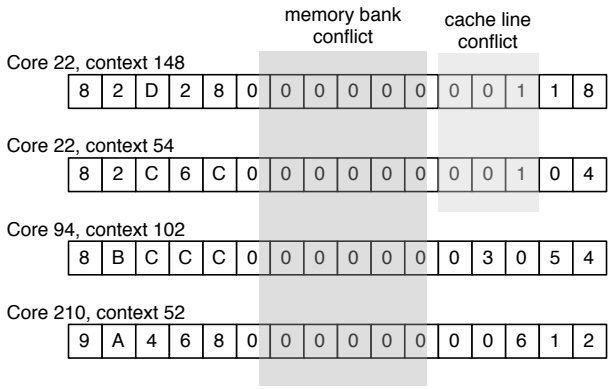


Figure 9.1: Potential bank and line conflicts with 64-bit addresses, 1024 cores, 256 contexts per core.

entire capacity of contemporary physical RAM chips. With 32-bit addressing however, this scheme is restrictive except with very few thread contexts overall.

While this scheme is transparent to use in programs, we found three significant issues at the level of system integration: *cache and bank conflicts*, *Translation Lookaside Buffer (TLB) pressure* and *storage reclamation*.

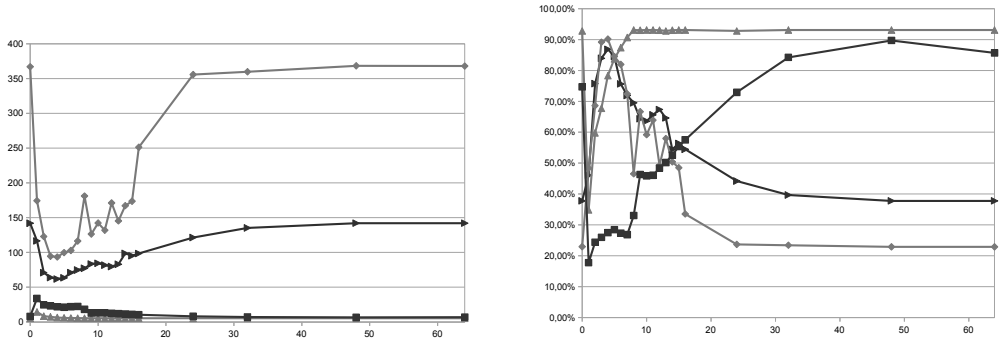
9.4.2.1 Cache and bank conflicts

Since most threads use only a small part of their TLS area, from the perspective of the memory system, most accesses are made to addresses that differ only via their most and least significant bits. Meanwhile, most common designs for bank and cache line selection in hardware select banks based on a *tag* computed with the middle bits of the address, which will be mostly identical between all threads in our scheme (fig. 9.1).

This effect was recognized in previous work [MS09], but existing solutions to avoid this situation are mostly applicable to external TLS distribution schemes, e.g. by randomization of the TLS base address for each thread context. This cannot be used here since we use computed distributions.

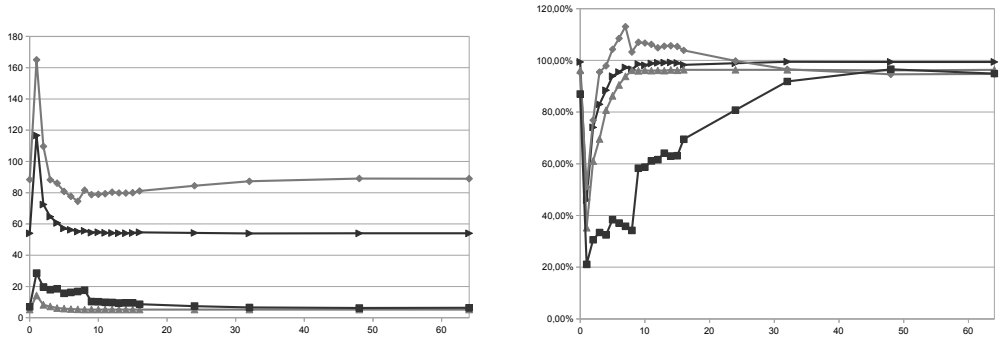
Instead, cache and bank selection randomization techniques do apply and are effective at mitigating this effect [RH90, Rau91, Sez93, GVTP97, KISL04, VD05].

To test this effect we have used four synthetic multithreaded benchmarks, two which do not use TLS at all and two that use less than 512 bytes of TLS per thread. The two programs



(a) Per-thread latency (clock cycles) (b) Instructions per cycle

Figure 9.2: Effect of TLS address conflicts with direct mapping.



(a) Per-thread latency (clock cycles) (b) Instructions per cycle

Figure 9.3: Reduced TLS address conflicts with XOR-based randomization.

which do not use TLS perform simple data-parallel computations. The two programs using TLS perform a more complex data-parallel operation which requires several procedure calls, and thus multiple stack-related operations besides the computation. We then ran these programs on a single multithreaded core with various numbers of thread contexts, given in the figures' x-axes. The processor is connected to a 128KiB 4-way associative L2-cache, which fits the working set of all 4 benchmarks.

As can be seen in fig. 9.2, the pattern of memory accesses in the two latter benchmarks incur a per-thread latency of several hundred cycles, contrasting with less than 50 cycles per thread for the simple benchmarks. Despite the potential for latency tolerance offered by fine-grained hardware multithreading, the number of instructions per cycle remains low (<50%, fig. 9.2b). Monitoring of the L2 cache activity reveals that more than 40% of accesses are misses caused by conflicts.

We then implemented an XOR reduction of the top and lower address bits to map L2 cache lines, inspired from [GVTP97]. As can be seen in fig. 9.3, this simple change caused a 400% performance increase of the most memory-bound benchmark, mostly by allowing independent memory accesses by different threads and thereby increasing IPC (fig. 9.3b).

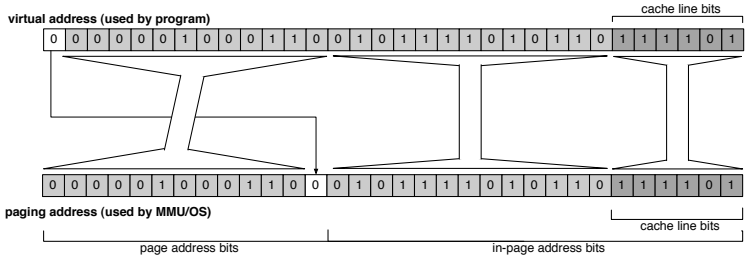


Figure 9.4: Address bit shuffling for non-TLS pages.

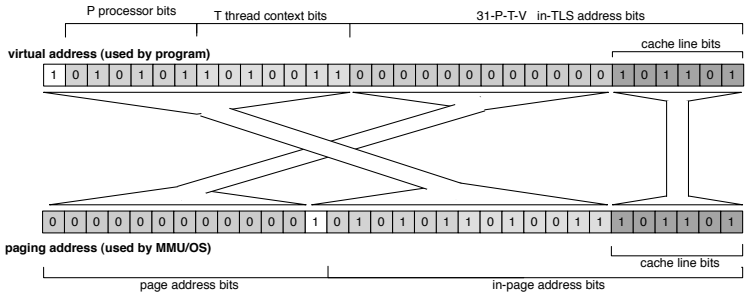


Figure 9.5: Address bit shuffling to aggregate distinct small TLS spaces into shared physical pages.

Note that the IPC measurement anomaly in fig. 9.3b is caused by an imprecision in our accounting: some cycles attributed to one benchmark, which causes its IPC to exceed 100%, should be accounted for another instead.

▶ To summarize, although an extensive study of cache design was not within the scope of our research, our work highlights that this TLS scheme places a strong need for tag randomization in the cache system.

9.4.2.2 TLB pressure

With a naive implementation of virtual addressing, our scheme would require at least one different translation entry for every thread context, because the address spaces of each TLS area (megabytes for 32-bit addressing, petabytes for 64-bit addressing, cf. table 9.3) would be larger than the virtual addressing page size (typically kilobytes on current systems). With large numbers of processors and thread contexts (e.g. 256×1024 contexts on large chips), this would require inefficiently large TLB implementations.

□ However, considering that the actual required TLS space in each thread is typically smaller than the page size, e.g. a few hundred bytes if only a few activation records are defined, we can propose an optimization to *aggregate* the actually used regions of TLS within single translation entries. To do this, we organize virtual addressing as follows.

We consider that there are 2^P cores (e.g. P = 6, 64 cores) and 2^T thread contexts per core (e.g. C = 7, 128 contexts). We also consider that there are 2^C bytes in a cache line (e.g. C = 6, 64 bytes). As a first step, we require that the virtual page size is at least 2^{C+P+T} bytes wide (e.g. 512KiB in our example). We consider that each page contains 2^B bytes,

Number of processors	Contexts per processor	Page size	TLS storage per thread in each page
8	8	4 KiB	64 B
8	8	16 KiB	256 B
8	8	1 MiB	16 KiB
64	64	256 KiB	64 B
64	64	1 MiB	256 B
64	64	16 MiB	4 KiB
1024	256	16 MiB	64 B
1024	256	128 MiB	512 B
1024	256	1 GiB	4 KiB
1024	1024	64 MiB	64 B
1024	1024	1 GiB	1 KiB
1024	1024	16 GiB	16 KiB

Table 9.4: Amount of TLS storage provided for each thread per virtual page.

Values assuming 64-byte cache lines.

with $B \geq C + P + T$. Then we modify the Memory Management Unit (MMU) as follows. When a request to translate a virtual address enters the MMU, the highest address bit is tested. If the bit is set, indicating TLS, the virtual address is first translated as depicted in fig. 9.5:

1. the lowest $B - P - T$ bits are left unchanged;
2. the $P + T$ most significant bits (except the highest) are shifted next to the lowest $B - P - T$ bits forming a complete in-page address;
3. the most significant bit (set) is shifted as the least significant bit of the page address;
4. the remaining original bits (from position $B - P - T + 1$ to $B - 1$) are shifted to the remaining positions of the page address (from $B + 2$ onwards).

If the highest bit is *not* set, then the address is first translated as depicted in fig. 9.4:

1. the lowest B bits are left unchanged;
2. the most significant bit (unset) is shifted as the least significant bit of the page address;
3. the remaining original bits (from position $B + 1$ onwards) are shifted to the remaining positions of the page address (from $B + 2$ onwards).

(The lowest C address bits are left unchanged to preserve locality within cache lines.) Then the resulting page address is looked up in the TLB and/or translation table(s).

With this scheme in place, each virtual page with an odd address provides 2^{B-T-P} bytes of storage to every thread context in the system (e.g. 64 bytes with 512KiB pages, 2KiB with 16MiB pages). This way, the number of TLB entries grows only as a function of the largest TLS address effectively used in the system. Since it is uncommon to see a large number of threads using simultaneously large TLS addresses, this relieves pressure on the TLB.

We provide a few examples of the provision for TLS per virtual page in table 9.4. As can be seen in this table, usual page sizes on contemporary operating systems (4KiB to 16MiB) are sufficient to provision at least a cache line per thread context up to moderately sized system (64 cores), and only a four-fold increase to at least 64 MiB per page is sufficient to support future-generation chips with thousands of cores.

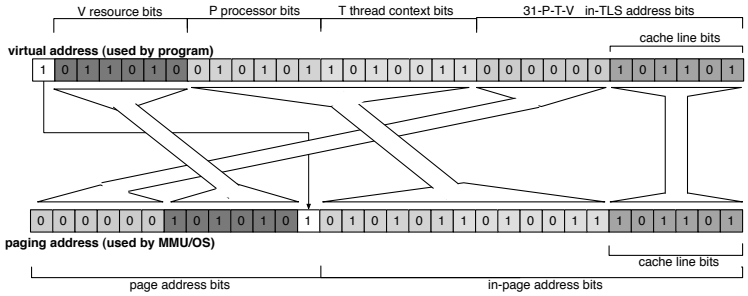


Figure 9.6: Address bit shuffling for non-TLS pages (virtual processes).

9.4.2.3 Reclamation of TLS storage

To analyze the issues related to reclamation, we must separately consider the situation without the TLB aggregation scheme presented above, and the situation once it is enabled.

Without the optimization, the issues of reclamation for this scheme are similar to those from the persistent dynamic allocation scheme described in section 9.3.1. The main difference is that the TLS address ranges are defined as translation entries in the virtual addressing subsystem, instead of per-context base addresses on each processor.

This makes aggressive reclamation difficult to implement: since the program and the virtual addressing subsystem are typically isolated from each other, the program must signal to an operating system every bulk creation and termination event to enable reclamation. Since the overhead of interactions with system services is typically higher than local procedure calls, it would defeat the entire approach and make persistent dynamic allocation more attractive. Context-based partitioning with deferred allocation and aggressive reclamation would only be viable if virtual addressing is under direct control of the program.

Delayed reclamation, in contrast, is tractable: a concurrent GC can asynchronously inspect the areas of memory pointed to by the active translation entries, and unmap all the unused areas. Or, alternatively, an operating system can release all translation entries at once when the program terminates.

With the aggregation optimization presented in the previous section, there is a significant problem however: *reclamation becomes impossible because each page contains a part of the TLS area of all thread contexts in the system* due to the bit shuffling.

□ To overcome this obstacle, we introduce *virtual resource identifiers* as part of the addressing scheme in addition to the physical processor identifiers, as depicted in fig. 9.6. These are numerical identifiers that delimit a memory management domain, for example a process in an operating system. In the addressing scheme, each access to TLS by a thread places the virtual resource identifier above the processor bits in virtual addresses, and the MMU shifts them to form the Least Significant Bit (LSB) of page addresses prior to address translation.

We suggest that this extension only applies to implementations with 64-bit addressing, since the size of the virtual resource identifier constrains the address width usable to address memory. Also, the width of the virtual address identifier in bits directly impacts TLB pressure, since TLS addresses from threads with different resource identifiers will map to different pages. This is similar to how synonyms increase cache pressure in virtually tagged caches [CD97a, CD97b].

However, virtual resource identifiers solve the reclamation problem. Indeed, since the resource identifier is part of the page identifier, two application components using distinct resource identifiers will map to different pages, and reclamation becomes possible by reclaiming entire sets of pages sharing the same resource identifier. This assumes that applications are constrained to avoid sharing of memory between program components that have distinct resource identifiers, or that explicit registration of common memory objects to a reclamation system service is mandatory.

▷ Final reclamation at program termination is the reclamation strategy we adopted in our implementation. We expect concurrent GC of translation entries to be better suited against over-allocation in long-running programs, however we could not explore this direction due to our limited effort budget.

9.5 Integration with the machine interface

Depending on the selected TLS scheme for a given implementation, the required architecture support differs:

- for dynamic pre-allocation and self-allocation, some form of *mutual exclusion* is required;
- for persistent dynamic allocation, both mutual exclusion and *access to a per-context TLS base pointer and size* is required;
- for all context-based partitioning schemes using external distribution, mutual exclusion may not be required, but access to the per-context TLS pointer and TLS size is required;
- for all context-based partitioning schemes using computed distribution, mutual exclusion may not be required but *access to a common TLS base pointer* and *TLS address space size* are required to compute the actual local, per-context TLS base pointer.

To summarize, either mutual exclusion or a way to obtain or configure the per-context TLS base pointer and size is required, or both. Since mutual exclusion can be introduced by traditional means (either memory-based atomics or as an operating system service), we focus here on the latter.

The requirement is for each logical thread to be able to retrieve the TLS base pointer for its own thread context, as well as the corresponding TLS size. Alternatively, the thread can be allowed to retrieve the first and last address of its TLS area, from which it can compute the size if needed.

In our work, we opted for the following extension to the machine interface:

- two new operations “ldbp” and “ldfp” are introduced, with the corresponding assembly mnemonics “`ldbp`” and “`ldfp`.” The first loads the base TLS address into its target operand, and the second loads the first address past the end of the TLS area. This allows us to both derive the size of the TLS and let a system implementation choose for upwards or downwards stack growth;
- in addition, two new operations “gettid” and “getcid” are introduced, with the corresponding mnemonics “`gettid`” and “`getcid`.” The first loads the local thread context identifier of the issuing thread into its target operand, and the second loads the local processor (core) identifier. Both use 0-based indexing. This allows us to implement context-based TLS schemes using explicit distribution with arbitrary data structures in memory.

```
1 mov    (__first_tls_top-__tls_base), %tl1  ! tl1 := sz
2 gettid %tlsp                               ! sp := tid
3 sll    %tlsp, %tl1, %tlsp                  ! sp := tid * sz
4 sethi  %hi(__first_tls_top), %tl1          ! tl1 := first top
5 add    %tlsp, %tl1, $tlsp ! sp := first top + tid * sz
```

Listing 9.1: Code sequence to access TLS on UTLEON3.

9.5.1 Example with the 1-core FPGA prototype

On the UTLEON3-based FPGA prototype implementation, we used the static/computed scheme, with 1 KiB bytes of RAM allocated to each of the 128 thread contexts.

To implement this, a 128 KiB array is defined in the program data. A linker symbol named "`__first_tls_top`" is placed at the address 1 KiB past the start of the array in memory. The compiler then emits the sequence in listing 9.1 at the start of every thread program which uses TLS. At run-time, each of these instructions executes in one processor cycle and access to TLS is gained in 5 cycles. It is possible to reduce this to 4 cycles if we fix the size to a static value, in which case the first operation is not needed and the size can be given as immediate constant to "`sll`."

9.5.2 Example with the multi-core system emulation

On the MGSim system emulation, we used the deferred/computed scheme. Here the implementation is simpler: the compiler simply needs to emit a single use of "`ldfp $sp`" at the start of every thread program which uses TLS. At run-time, the bits required to construct the TLS final address are available in the pipeline and allow the operation to complete in 1 processor cycle. The size of the individual TLS areas can be further obtained by subtracting the output of "ldbp" from the output of "ldfp."

Protocol	Mechanism	Time cost†
Dynamic pre-allocation	procedure call or active message by creating thread	A_1
Self-allocation	procedure call or active message by each created logical thread	$A'_1 \times N$
Persistent dynamic allocation	procedure call or active message by the first logical thread of a context	$A''_1 \times P$
Context-based pre-reservation	static pre-allocation, external distribution	$A_1 + A_2 + C_1(P)$
Context-based pre-reservation	deferred pre-allocation, external distribution	$A_2 + C_1(P) + C_2(P)$
Context-based pre-reservation	static pre-allocation, computed distribution	$A_1 + C_3(N)$
Context-based pre-reservation	deferred pre-allocation, computed distribution	$C_4(P, N)$

† N: number of logical threads created. P: number of thread contexts participating in the bulk creation. A_1: cost to perform one allocation of TLS space. A_2: cost to allocate an array of pointers. $C_1(P)$: cost to compute and store one pointer for each context. $C_2(P)$: cost to provision virtual memory upon first access by each context (may be aggregated across multiple contexts and occur concurrently). $C_3(N)$: cost to compute one pointer in each logical thread (may occur concurrently). $C_4(P, N)$: combined cost of provisioning virtual memory and computing pointers (may occur concurrently).

Table 9.5: Protocols to provision TLS for one bulk creation.

Summary

Provisioning Thread-Local Storage (TLS) to threads is an essential feature of a general-purpose system, mostly because per-thread stacks are necessary to support recursion and arbitrary large numbers of local variables.

One specific challenge related to massively concurrent architectures is support for a large number of threads and the design objective to keep thread creation cheap throughout the abstraction stack. This opposes traditional operating software approaches which exploit a common memory allocation service using mutual exclusion, as this would become a contention point as the rate of thread creation increases overall.

Another challenge related to the specific design from part I is that TLS is a feature of logical threads, whereas our physical unit of concurrency is the physical thread context. As the number of logical threads can be arbitrary large but they are serialized over the hardware contexts, TLS should be allocated per context and not per logical thread.

□ In this chapter we have reviewed traditional approaches to provision TLS to threads. We summarize the various options in table 9.5. We presented multiple schemes that pre-allocate address space and associate semi-statically TLS areas to thread contexts. They address both ▶▷ challenges identified above at once. They involve joint support from the architecture and from operating software providers, in particular with regards to *reclamation* of unused TLS ranges.

Chapter 10

Concurrency virtualization
—Highlight on specifics, opus 3

Abstract

A key role of abstraction is to hide the finiteness of physical-world resources, in particular through their *virtualization*. Virtualization in turn requires specific support from the underlying architecture, and thus constitutes an ongoing interaction between the platform provider and operating software provider. In this chapter, we illustrate this interaction for the virtualization of the "family," a group of cooperatively scheduled logical threads that can be created and joined using single bulk operations.

Contents

10.1 Introduction

As noted in [Day11], a duality exists between specialization and generalization when constructing abstractions. "Specialization" refers to language restrictions, compile-time (static) solutions and the exploitation of machine-specific facilities—in the interest of efficiency. "Generalization" refers to general language constructs, run-time (dynamic) solutions and machine-independent language design—in the interest of correctness and reliability. When catering to operating software providers, the hardware architect must recognize this duality and enter a dialogue at both levels.

In particular, general abstractions often hide implementation-specific resource limits, to ensure portability and forward compatibility of software. This hiding is achieved by introducing "virtual" resources, which appear unbounded in the abstract semantics of programming languages. *Resource virtualization* thus achieves a separation of concerns between software specification and its realization at run-time where resource constraints apply. To implement resource virtualization, platform providers and operating software providers must co-design *virtualization mechanisms*. For example, the virtualization of variable slots in programming languages is typically supported by introducing relative addressing in processors and offset tables in compilers. Meanwhile, the need for specialization requires that users of abstractions can control resources explicitly *when so desired* [Kic91]. Any virtualization mechanism should thus provide and document optional control to disable or bypass virtualization.

In the context of general-purpose systems, the need for resource virtualization has traditionally been offset by ensuring that most resource abstractions available in programing languages could be expressed in terms of data structures in memory: once multiple resources can be represented by memory, only one virtualization mechanism for memory is needed to virtualize these resources. This is the approach taken e.g. in Unix-like operating systems where inter-process communication channels, files and synchronization devices are all represented by data structures in memory and thus appear unbounded with the backing of virtualized memory. Processes, which are, resource-wise, a combination of virtualized processors and memory, require a separate mechanism to virtualize processors; Unix uses time sharing for this purpose.

When innovating in processor chip design, it is thus necessary to interact with the operating software providers to define virtualization mechanisms for all the resource types introduced. As the Unix experience illustrates, a potential strategy is to represent new resource types in terms of data structures in virtual memory.

10.1.1 Example new resources to virtualize: concurrency contexts

As we explained in part I, the proposed architecture design introduces "execution contexts" for concurrent tasks, dedicated to optimize the management of concurrency throughout the system stack. The architecture design introduces three types of resources: one is the *logical thread of execution*, another is the *bulk creation context* to prepare the automatic creation of multiple logical threads, and the last is the *bulk synchronizer* to synchronize on the termination of multiple logical threads at once.

As explained in section 3.3.1, the proposed machine interface already provides dedicated support to virtualize logical threads: it states that an arbitrarily large number of logical threads are automatically scheduled over the physical thread contexts. Thread virtualization is thus achieved in hardware.

Meanwhile, our proposed abstract machine (chapter 7) defines primitives to create and synchronize on concurrent tasks hierarchically as abstract *families*, at run-time. Its semantics make an assumption of unlimited hardware resources, by implying that creating logical threads and waiting on their termination always succeeds. In other words, our abstract machine communicates the illusion that programs can always create new families, regardless of where in the program, and when during run-time, creation is expressed. This enables resource-agnostic programming where concurrent program blocks can be freely composed without having to choose which computation is defined as a concurrent family and which computation should be defined as a sequential process.

However, the bulk creation contexts and bulk synchronizers have an explicit, named existence in the machine interface, as explained in chapters 3 and 4. Without support for virtualization of the bulk creation context and bulk synchronizers, it is possible to express programs that may define more families during their execution than there are bulk creation contexts available in hardware. For example, divide-and-conquer algorithms exhibit this situation when the depth of the recursion is data-dependent. With a transparent mapping of the input language construct for family creation to machine primitives, progress and termination of otherwise semantically valid deterministic programs would not be guaranteed, since an extra context allocation when all hardware resources are busy would yield a deadlock or failure (section 4.3.1.1).

To resolve this mismatch, we propose in this chapter a controllable mechanism to virtualize the bulk creation contexts and bulk synchronizers. In section 10.2, we start by outlining how concurrency resources are managed and virtualized in related work. We then outline our proposed protocol in section 10.3 and analyze its limitations in section 10.4.

10.2 Context and related work

The example situation we are addressing here is averted in most existing concurrency management environments, in either of two ways:

- in *resource-aware* concurrent programming environments, the resource limitations of the underlying execution platform are exposed in the programming language semantics: the programmer or automatic program generator must either acknowledge and assume a fixed set of processing units known *a priori*, or explicitly query at run-time how many processing units are available before starting parallel execution.
  This situation can be found, for example, with programs that distribute work over parallel "worker" processes implemented with POSIX processes or threads; the application must query the number of actual processors in the system, and possibly also how many POSIX processes/threads are supported by the operating system, to determine how many workers to start. The Apache web server application[1] uses this scheme. It can also be found with the MPI environment, where a process needs to query the number of processes involved in a communicator with `MPI_Comm_size` before spawning sub-processes with `MPI_Comm_spawn`.
- in *virtualized resource-agnostic* concurrent programming environments, most implementations support unbounded numbers of *virtual concurrency contexts* backed up by main system memory: any request to spawn a new parallel thread of execution is matched by the dynamic allocation, in main memory, of the required data structures.

[1] `http://httpd.apache.org/`

In this situation, the maximum amount of concurrency that can be defined is typically orders of magnitude larger than the number of hardware processing units, ensuring that the limit is never reached in all practical circumstances.
This situation can be found in most existing task-based concurrency interfaces: tasks in .NET's Thread Parallelism Library (TPL) [Duf09] and Intel's Threading Building Blocks (TBB) [Rei07], blocks with GCD [App, Sir09], tasks with OpenMP [Ope08], etc.

In both situations, there is no interface mismatch: in the former case, the input concurrency matches the number of execution contexts by design, and in the latter case the number of execution contexts scales automatically with the amount of concurrency defined by the input program.

Arguably, resource-agnostic programming should be the privileged approach for programming current and future multi-core systems [PJ10]. However, virtualization of execution contexts, where there can be more execution contexts defined than there are processors available, often incurs implementation complexity and run-time overheads: unless some restriction is made on scheduling, space must be allocated from memory to save the state of tasks at every context switch. This incurs contention on memory allocators and extra traffic on the memory network.

An example restriction that alleviates the need for saving the state of all tasks to memory is so-called *declarative, resource-agnostic* concurrent programming environments. In these, programs declare concurrent operations to perform without indicating whether, where or how to perform them, and the language compiler and underlying platform then cooperate to determine the cheapest and most efficient execution strategy. The specific advantage of *declarative concurrency constructs* in the associated languages is that any excess concurrency can be serialized cooperatively, as a loop or recursion, automatically within single execution contexts without changes in semantics. As such it constitutes a form of *cooperative virtualization*.

This is the approach exploited by e.g. the automatic folding of concurrent spawns in the parent task in Cilk [BJK+95], the aggregation of multiple iterations of loops as single tasks in OpenMP [Ope08], the various aggregation strategies of the SAC2C compiler of Single-Assignment C [GS06], the automatic aggregation of parallel algorithms via "partitioners" by Intel's TBB run-time system [Int11], the automatic serialization of data parallel algorithms by the Chapel compiler and run-time system [Cra11].

10.3 Controllable automatic serialization

□ In this chapter, we propose a possible implementation of declarative concurrency for the proposed target architecture. While doing so, we remove partly from the language semantics the transparency between the concurrency expressed in the program and actual parallelism of execution on the platform at run-time. Our strategy is to delay the choice between concurrent execution, via hardware families, and sequential execution within the context of an existing active thread, via serialization, until run-time when the actual availability of execution units is known. Meanwhile, we reserve the opportunity to control the virtualization via optional language constructs. We thus propose two interfaces to control concurrency:

- when the *implicit form* of concurrency constructs is used in programs, the automatic scheme is used and run-time resource availability is used to choose between concurrent and serial execution;

- additionally, a programmer, code generator or run-time system implementer can add *explicit specifiers* to the constructs to either force concurrency creation or force sequentialization in a specific syntactic locus.

In both cases, the semantics presented in chapter 7 apply to describe the functional semantics of programs. With the implicit form, progress and completion of workloads are guaranteed by reusing existing contexts for any excess concurrency. With the explicit form, progress and completion are only guaranteed if the entity that wrote the program (programmer or code generator) collaborates with a resource management service on the system to not require more contexts than are available in the actual platform.

10.3.1 Implementation

Prior to the introduction of "soft failure reporting" in the allocation of bulk creation contexts (cf. section 4.3.1.1), we attempted to expose *resource counters* in the hardware that would reflect the occupancy of the hardware structures. We could make these counters visible either via special memory addresses or dedicated instructions in the ISA. However, we found that any counter-based scheme which does not also adapt the family allocation mechanism is flawed, because the fine-grained concurrency in the system makes the value of such counters essentially *inaccurate*: there can be family allocations, or de-allocation events, between a point a counter is sampled and the point the value is acted upon. Generally, any feedback mechanism must ensure that the test on availability and the allocation of a processing resource if there is one available are performed *atomically*.

■ Instead we co-designed "soft failure reporting" with the architecture implementers. With this feature the "allocate" primitive returns a special "invalid" value upon failure to allocate a bulk creation context. We further introduced a parameter to the operation, to select whether failure allocations are reported to the software ("soft failure"), or whether allocation should wait indefinitely until a context becomes available ("suspending"). The latter option allows to implement mandatory concurrency, or mandatory delegation of an activity such as needed for system services (section 5.5.1), without using busy waiting loops on allocation failures.

Once the feedback mechanism becomes available in the target implementation, we can use it during code generation as follows: whenever execution reaches a point where a concurrent family of threads is *declared* in the input program, the executed code first attempts the context allocation[2]. Then, if the result indicates a failure, the code branches to the serialized version of the declared work. Otherwise the declared work is created as a concurrent family of logical threads.

To illustrate this scheme, we consider the example code from listing H.21, which defines a thread program "innerprod" and uses it in a "kernel" function. With our strategy enabled, the translation stage before code generation produces the code given in listing 10.1, which in turn causes code generation and post-processing to produce the assembly code given in listing 10.2 for the SPARC-based target ISA.

This example allows us to highlight the following:

- the scheme requires duplication of the code of thread programs: one version must be generated suitable for creation as a thread (listing 10.1, line 5), and another suitable for a procedure call in C (line 6); furthermore, both must be declared before they are defined, in case the input program uses recursion;

[2]This causes the hardware allocator to attempt an allocation of the desired number of contexts, or any smaller size down to a single context, or fail if no context is available whatsoever.

```
1  void _mts_innerprod(void);
2  void _seq_innerprod(long _I, int *, int *, int *);
3  extern void * innerprod[2];
4
5  void _mts_innerprod(void) { /* thread program */ }
6  void _seq_innerprod(long _I, int* _mtp_a, int* _mtp_b,
7      int * _mtp_sum) { /* plain C function */ }
8  void * innerprod[2] =
9  { &_mts_innerprod, &_seq_innerprod };
10
11 int A[100];
12 int B[100];
13
14 sl_def(kernel,, sl_shparm(int, res)) {
15   long _fid, _start, _limit, _step, _block, _idx;
16   int * _arg1, * _arg2, _arg_sum_11;
17
18   /* common initializers from program source: */
19   _start = 0, _limit = 100, _step = 1, _block = 0;
20   _arg1 = A,  _arg2 = B, _arg_sum = 0;
21
22   /* allocation test: */
23   asm volatile("allocate␣%0" : "=r"( _fid));
24   if (-1 == _fid)
25     goto _seq;
26
27   /* here configure and create as family of threads,
28      using context _fid,
29      range (_start, _limit, _step),
30      window size _block,
31      source values _arg1, _arg2, _arg_sum,
32      thread program _mts_innerprod,
33      then synchronize and update _arg_sum */
34   goto _end ;
35
36 _seq:
37   if (_step > 0)
38     for (_idx = _start; _idx < _limit; _idx += _step)
39       _seq_innerprod(_idx, _arg1, _arg2, &_arg_sum);
40   else
41     for (_idx = _start; _idx > _limit; _idx += _step)
42       _seq_innerprod(_idx, _arg1, _arg2, &_arg_sum);
43
44 _end:
45
46   sl_setp(res, 0);
47 }
```

Listing 10.1: Automatic serialization code for listing H.21 (simplified).

```
1  ! (``innerprod'' code omitted)
2      .global _mts_kernel
3      .type   _mts_kernel, #function
4      .registers      0 1 6 0 0 0
5  _mts_kernel:
6      allocate %tl3                !. ``try or fail''
7      cmp      %tl3, -1; swch      !. success?
8      be       .LL13               !. no: branch below
9      nop
10     sethi    %hi(A), %tl0        !+ load channel source
11     or       %tl0, %lo(A), %tl0  !+ load channel source
12     sethi    %hi(B), %tl1        !+ load channel source
13     or       %tl1, %lo(B), %tl1  !+ load channel source
14     mov      0, %tl2             !+ load channel source
15     setlimit %tl3, 99            !+ configure limit
16     sethi    %hi(_mts_innerprod), %tl5  !+ load PC
17     or       %tl5, %lo(_mts_innerprod), %tl5
18     setthread %tl3, %tl5         !+ configure PC
19     create   %tl3, %tl3; swch    !+ create family
20     mov      %tl3, %tl3; swch    !+ wait on termination
21 .LL14:
22     mov      %tl2, %ts0          !. propagate result
23     end                          !. end thread
24 .LL13:
25     sethi    %hi(B), %tl4        !- load argument
26     sethi    %hi(A), %tl5        !- load argument
27     mov      0, %tl0             !- init counter
28     mov      0, %tl2             !- load argument
29     or       %tl4, %lo(B), %tl4  !- load argument
30     or       %tl5, %lo(A), %tl5  !- load argument
31 .LL15:
32     ld       [%tl4+%tl0], %tl3       ! inlined load
33     ld       [%tl5+%tl0], %tl1       ! inlined load
34     add      %tl0, 4, %tl0           !- increment counter
35     smul     %tl3, %tl1, %tl1; swch ! inlined mul
36     cmp      %tl0, 400               !- test counter
37     add      %tl2, %tl1, %tl2; swch ! inlined reduction
38     bne      .LL15                   !- next iteration
39     nop
40     b,a      .LL14               !- back to common end
41     nop
42     .size    _mts_kernel, .-_mts_kernel
```

Listing 10.2: Generated assembly code for listing 10.1.

- once the definition of the C function is in scope, the inlining capabilities of the code generator can operate as usual; this is demonstrated in the example where the code generator inlines the kernel loop entirely (listing 10.2, line 31);
- although there are two loops to cater for both increasing and decreasing index sequences, usually constant propagation of the step value by the code generator will ensure that only one branch is compiled, as demonstrated in the example;
- the overhead of the scheme compared to a pure sequential code using a loop is 3 machine instructions: one use of family allocation (listing 10.2, line 6) and a conditional branch (lines 7 and 8);
- the overhead of the scheme compared to a transparent use of the concurrency primitives of the underlying architecture is 2 machine instructions, i.e. the conditional branch;
- although the two versions can be separately identified by name mangling[3], we also need to generate a single *function descriptor* (listing 10.1, line 8), defined as an array of pointers to the two codes and with the original name "innerprod," to allow a single address for the function to be taken and stored in C data structures, as required for system code;
- although an allocation failure causes a serialized version of the family-defining loop to be used, this does not imply that the family is fully serialized: if the thread program of the serialized family uses further family creations, these will attempt to allocate a concurrent context again. This allows parallelism to be re-gained dynamically once some contexts become available.

10.3.2 Impact on the language semantics

Thanks to the requirement of serializability introduced early on (section 6.2.4), the base language semantics as of chapter 7 and Appendix I did not require any change with the introduction of this automated resource scheduling scheme in the *implicit* form of the concurrency constructs. In particular, all our definitions about scheduling, ordering of side effects, etc. discussed in chapter 7 apply to the new operational semantics of our implementation just as well as they did without the scheme implemented.

□ However, as described in section 10.3, we also found necessary to cater for the need of system code by providing an *explicit* control over this scheme. We did this by extending the syntax form of *create specifiers* (Appendix I.5.8.1):

create-specifier:
...
`sl__forceseq`
`sl__forcewait`

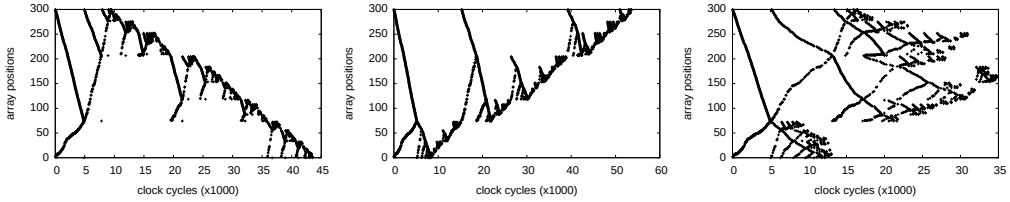
With this syntax extension, it becomes possible to use the words "`sl__forceseq`" or "`sl__forcewait`" as the 6th positional parameter of the "`sl_create`" construct. The code translator can then determine which allocation mode to use, respectively "wait and succeed" or "try or fail," and also avoid emitting the version of the creation code that is left unused (respectively the serialized and non-serialized part). See also side note 10.1.

From a semantics perspective, these specifiers do not modify the functional behavior of the program; however we intentionally choose to not *advertise* their existence in our

[3] e.g. "`_mts_innerprod`" vs. "`_seq_innerprod`"

Side note 10.1: Static vs. dynamic allocation failure management.

The syntax-based discrimination is *static*: the specifiers are lexical keywords and it is their presence or absence in the syntax that determines the choice at compile-time.
We chose a static scheme over a dynamic scheme, i.e. where the run-time value of an expression would determine the behavior, because we found that in all situations where control is desired, we know statically which implementation should be used. If future work shows that a dynamic choice is required instead, we estimate that the additional implementation work required to add a run-time conditional in the generated code is minimal; however we highlight that a run-time conditional would prevent the static elision of whichever version is not used from the generated code.



(a) Sequential (b) 1 family context (c) 32 family contexts

Figure 10.1: Execution of QuickSort on 1 core.

These diagram represent memory activity over time. The horizontal axis represents time; the vertical axis represents cells in the array being sorted. A dot in the diagram represents a memory read or write operation. In a sequential execution, only one branch of the recursive divide-and-conquer step is executed at a time (e.g. figs. 10.1a and 10.1b); with concurrency, multiple branches are executed simultaneously (e.g. fig. 10.1c).

published interface language specification, since they require the conceptual introduction of a resource model which does not otherwise exist in C.

10.3.3 Run-time behavior

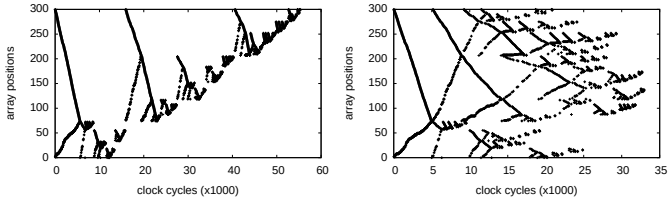
In Appendix J we detail the behavior of two naive recursive QuickSort implementations with our strategy enabled.

As expected, the use of our concurrency constructs adds overhead compared to a pure sequential version if the program is forced to run as a single thread, because the hardware must stall the execution at every attempt to allocate a bulk creation context until the allocation unit reports a failure. However, as soon as more than one context is available in hardware, it is automatically exploited by the program and the resulting multithreaded execution provides latency tolerance and speedup, even on one core (fig. 10.1).

Interestingly, we found that it is not very beneficial to instrument the naive version to only use concurrency when the amount of work at a given recursion level, here the size of the sub-array to sort, is larger than a threshold (fig. 10.2). Indeed, since the cost of testing the sub-problem size is comparable to the cost of trying an allocation, any threshold-based alteration to the scheme amounts to trading one time overhead with the same.

10.4 Limitations and future work

As seen above, our scheme incurs a check overhead, with longer execution times than a pure sequential program, if concurrent code executes in a mostly sequential environment.



(a) 1 family context (b) 32 family contexts

Figure 10.2: Execution of QuickSort on 1 core (threshold: 16 elements).

Furthermore, this overhead cannot be trivially avoided by means of a threshold on the problem size, because simple conditionals have a comparable cost to a failed allocation.

We generalize by stating that a program containing dynamically heterogeneous concurrency should only be expressed using concurrency constructs in the source code if a reasonable confidence exists that multiple family contexts will be available at run-time to provide multithreading overlap and compensate for the overheads. This confidence can be provided by an on-chip resource manager that partitions the chip between application components (cf. chapter 11). Otherwise, the concurrency management overheads contribute to the critical path through the execution, as highlighted in [FLR98].

▷ We suggest that the scheme could be extended by duplicating the entire recursion as a purely sequential program, with a unique start-up check in the algorithm entry point on whether the underlying execution resource is sequential or not. In the sequential case, the sequential code is used and this overhead could be avoided. This possible extension constitutes future work.

▷ There is another limitation that can be observed when running families defining many logical threads, for example one thread per item in an array. If an algorithm exhibits both dynamically heterogeneous concurrency, e.g. data-dependent divide-and-conquer, and large families, it becomes possible that the serialization of a family causes a load imbalance: once a leaf family starts to run as a loop, it will execute entirely as a loop, even if concurrency contexts become available during the loop execution. We suggest to extend our scheme to try and allocate contexts within the loop body, and switch back to parallel execution when a context becomes available. To avoid a detrimental effect on the performance of inner loops, these attempts should be performed by an outer loop with a configurable coarser step than the inner loop. This should be explored in future work.

▷ Furthermore, while working on TLS (chapter 9), we discovered another limitation to this scheme: the serialization of a family of threads as a loop also merges any thread-local storage requirements, e.g. minimum stack frame sizes, of the serialized family into the thread where the serialization occurs. In other words, in any situation where serialization *may* occur, the execution environment *must* provision extended thread-local storage to each serializing thread context. With recursion, the thread-local storage of some threads may become arbitrarily large during execution. This in turn requires an efficient, low-latency dynamic memory allocation scheme with extremely low contention to keep all threads on each core active [Gru94]. Concurrent dynamic memory management is a known active field of research, and we expect any future result in that direction to be relevant to the architecture considered.

▷ Finally, we highlight that the virtualization scheme described in this chapter is cooperative. In particular, it requires that the programming language semantics do not expose *mandatory parallelism*, i.e. the ability for a program description to require from the platform that two activities must be carried out in parallel (e.g. so that they can communicate bidirectionally). This requirement holds for languages like our proposed interface from chapters 6 and 7, where the main constructs for concurrency are declarative. However, as we suggested in section 6.3.6 and detail further in chapter 12, other programming interfaces may be devised for the platform. For example an implementation of Unix would require another form of concurrency virtualization that guarantees fair scheduling between independent processes. If or when this happens, and an alternate interface offers mandatory parallelism in its semantics, other virtualization mechanisms will be required.

Summary

□ In this chapter, we have highlighted the dialogue between the hardware architect and the operating software provider to design *virtualization mechanisms* that abstract away the finiteness of physical resources. This dialogue must be engaged each time a new resource type is added to a platform design.

We have illustrated this dialogue with the example of *concurrency resources* in the proposed architecture, which are finite on any given implementation and are desirably unbounded ■□ in abstract models. We have addressed this virtualization need by introducing *declarative concurrency constructs* in the interface language. We have also proposed an implementation of these constructs that automatically and cooperatively sequentializes units of work at run- ▷ time when concurrency resources in the underlying platform become exhausted. We have isolated a need to manage the mapping of workloads to resources in future work. We have also highlighted that declarative concurrency may not be suitable for other programming abstractions, where different virtualization mechanisms may be required instead.

Chapter 11

Placement and platform partitioning
—Highlight on specifics, opus 4

> The way forward seems to be to provide reasonable operations, and make costs somewhat transparent.
>
> Andrei Matei

Abstract

When adding new architectural features that do not have corresponding abstractions in the operating software stack, it is sometimes necessary to create "holes" in the lower level interfaces so that the new features can be exploited directly. However the constructs to drive these new features should still be designed to abstract away uninteresting characteristics of the implementation, and expose instead its general principles. We illustrate this with the example of *work placement*. As explained by [PHA10] and previous work, placement optimization cannot be fully automated in hardware and the efficient execution of fine-grained concurrent workloads over large many-core chips will require dedicated interfaces to control the mapping of computations to specific cores in operating software. However, no consensus yet exists as to what these interfaces should be. Upcoming many-core system stacks will thus need to explore placement issues by exploiting primitives that are not yet part of the usual abstraction tool box. In this chapter, we discuss how such primitives from the architecture proposed in part I can be exposed to the operating software providers.

Contents

11.1 Introduction

In chapter 5, we outlined the special relationship between platform providers and the providers of "first level" operating software, such as the C programming environment. We also highlighted the integration process for innovations at the level of the machine interface: first level operating software is first extended beyond the scope of established standards, then applications start to use non-standard features, and then features confirmed to be useful across a range of applications are retrospectively integrated in new revisions of the standards. To enable this process, there must exist "holes" in the abstraction layers to provide direct access to new features. An example of this is the `asm` construct in most C implementations, which allows C programs to use machine instructions not yet used by existing C language features.

Placement of computations and data on physical resources on a chip is another example of a platform feature still missing standard abstractions.

In contemporary systems, explicit placement is usually available through a system API with a large granularity: a program wishing to *bind* a computation, usually a thread or a process, to a particular processor will request so through an interface to the operating system's scheduler. Examples are Linux' `sched_setaffinity` or FreeBSD's `cpuset_setaffinity`. This usually requires special privileges and the overhead of a system call, and in practice is usually only performed once at program start-up to assign "worker" threads to specific processors, which then cooperatively negotiate the dispatch of tasks through data structures in shared memory. Even when considering the task dispatching process itself as a placement mechanism, overheads are also incurred by the cache coherency required to communicate the task data structures across cores in the memory network.

In contrast, the concurrency management primitives available with the architecture from part I allow programs to optionally and explicitly control the mapping of computations onto cores via a dedicated "*delegation*" NoC. The delegation messages are issued via special ISA instructions. This protocol enables the dispatch and synchronization of multi-core workloads within a few pipeline cycles; the direct use of the corresponding ISA instructions by programs is therefore orders of magnitude faster than the overhead of calling a system API, and *also* faster than sharing task data structures via a memory network.

However, direct use in programs also implies that the first level interface languages in the operating software must expose the hardware primitives as first-class language constructs that can be used at any point in algorithms. In the case of C, for example, this is a radically new requirement as C had never exposed knowledge about the physical location of sub-computations previously.

What should this language interface look like? One the one hand, we can predict that a first level interface can hide hardware-specific details such as the encoding of the special instructions. On the other hand, there is obviously not enough information at the lower level of the abstraction stack to devise high-level abstractions, for example "automatically perform the best work distribution for a given parallel computation." This knowledge and the corresponding decision mechanisms are typically found at a higher level in the operating software stack, for example in the run-time system or the user of a "productivity" language like Chapel [CCZ07] or SAC [GS06].

This situation thus warrants creating an "abstraction hole," to delegate to operating software providers at a higher abstraction level the responsibility to make placement decisions. In this chapter, we illustrate how to achieve this with the proposed architecture in our interface language SL, previously introduced in chapter 6.

11.2 On-chip work placement: exposing control

11.2.1 Architectural support

○ The machine interface in chapter 4 was designed with two mechanisms for placement in mind.
Historically, the first mechanism was a design-time partitioning of the many-core chip into
■ *statically named clusters*. Later on, a flexible *dynamic partitioning protocol* was introduced as well, which we jointly designed and prototyped with the platform implementers. Both can be used in combination in a design, where larger, static cluster can be "internally" divided further at run-time into fine-grained sub-clusters. We detail the latter fine-grained placement protocol in Appendix E.

In both cases, the "allocate" instruction that reserves bulk creation contexts accepts a "placement" operand. This is a resource address which triggers remote access, through the delegation NoC, to the named cluster if the address does not match the *local address* of the thread issuing "allocate." With static clusters, the address corresponds to the master core connected both to the delegation NoC and its neighbour cores in the cluster; with dynamic clusters, a *virtual cluster address* is formed by the address of a first core, together with a cluster size and security capability, combined in a single integer value.

Jointly to this addressing mechanism for new delegations, two machine instructions "getpid" and "getcid" are introduced. "getpid" produces the address of the cluster where the current thread was delegated to, also called *default* or *current* placement. "getcid" produces the address of the local core within its cluster. To avoid the overhead of extra instructions, "allocate" also recognizes the symbolic value 0 to implicitly reuse the default placement; and it also recognizes 1 to use the local core only, i.e. avoid multi-core execution altogether.

With large static clusters, an operating software would likely not bind small computing activities to the static clusters at a high rate. In other words, the granularity of placement over large static clusters is coarse enough that this type of placement can be requested via system APIs. It is the existence of small clusters and the ability to dynamically partition clusters at run-time, within a few hardware cycles, which justifies the need for low-overhead, direct access via language primitives.

11.2.2 Language primitives

□ We propose to transparently pass the run-time value of the "place" family configuration expression, which is the 2nd positional parameter to the "`sl_create`" construct (clauses I.5.8.1§3 and I.5.8.1§26), as the placement operand of the "allocate" operation. If the place expression is not provided (it is optional in the input syntax), the value 0 is used. With this translation in place, a program can specify, at the point of family creation:

- either no place expression at all, which uses the default placement implicitly, or
- one of the symbolic values 0 or 1, which designate the default placement or the local core explicitly, or
- a fully specified target cluster address.

□ Separately, we extend the library environment as follows:

- we encapsulate the "`getpid`" and "`getcid`" machine instructions as C macros, defined in a SL library header, named "`sl_default_placement()`" and "`sl_local_processor_address()`," respectively;

Name	Description	Cost[1]
`sl_placement_size(C)`	Number of cores in the cluster C.	2
`sl_first_processor_address(C)`	Address the first core in the cluster C.	2
`at_core(P)`	Address the relatively numbered core P in the current cluster.	5/6
`split_upper()`	Split the current cluster and address the upper half.	5
`split_lower()`	Split the current cluster and address the lower half.	5
`split_sibling()`	Split the current cluster and address the sibling half from the perspective of the current core.	11
`next_core()`	Address the next core within the current cluster, with wraparound.	13
`prev_core()`	Address the previous core within the current cluster, with wraparound.	13

[1] Number of processor cycles required with the Alpha or SPARC ISAs.

Table 11.1: Core addressing operators in the SL library.

- we define a C typedef name "`sl_place_t`" wide enough to hold placement addresses.

With these primitives, a thread in the program can read and manipulate its own placement information, for example to compute thread distributions explicitly.

11.2.3 Integer arithmetic to compute placements

A characteristic feature of the proposed protocol in Appendix E is that *it exposes the format of core addresses* on the delegation NoC. In contrast to most contemporary systems where knowledge about the inter-processor fabric is entirely encapsulated in system services queried via privileged APIs, the proposed protocol allows programs to construct processor addresses "from scratch" without involving a system service.

To achieve this, the protocol advertises that a valid virtual cluster address is constructed by concatenating together a security capability[1], the address of the first core in the cluster and the desired cluster size in a single machine word. This encoding creates the opportunity for programs to compute *relative core addresses* with the overhead of a few integer arithmetic instructions. For example, in our work we have captured the common patterns in table 11.1 as inlineable functions in a SL library header. Their implementation is given in Appendix E.3.

11.2.4 Example usage: inner product

The abstraction above allows programs to express *relative constraints* on placement of computations, useful to distribute reductions over variable numbers of cores.

[1]The capability is optional and can be used by a coarse-grained system service to provide isolation between computing activities running on separate large-scale core clusters on the chip.

```
1  double kernel3(size_t n, double *X, double *Z)
2  {
3    sl_create(,,, n, , , , innerk3,
4              sl_shfarg(double, Qr, 0.0),
5              sl_glarg(double*, , Z), sl_glarg(double*, , X));
6    sl_sync();
7    return sl_geta(Qr);
8  }
9
10 sl_def(innerk3,, sl_shfparm(double, Q),
11         sl_glparm(double*, Z), sl_glparm(double*, X))
12 {
13   sl_index(i);
14   // Q += Z[i] * X[i]
15   sl_setp(Q, sl_getp(Z)[i] * sl_getp(X)[i] + sl_getp(Q));
16 }
17 sl_enddef
```

Listing 11.1: Concurrent SL code for the Livermore loop 3 (inner product).

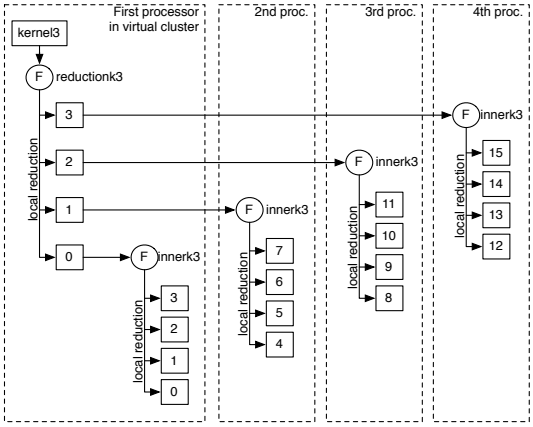


Figure 11.1: Parallel reduction using explicit placement, for problem size $n = 16$ and a virtual core cluster containing 4 cores.

The boxes indicate the generated logical thread indexes.

As an example, consider the naive implementation of the vector-vector product in listing 11.1. This defines a C function `kernel3` which computes the inner product of the n-sized vectors `X` and `Z` using a single family of threads. Because there is a carried dependency between all threads, this implementation cannot benefit from multi-core execution: the first logical thread on every core but the first will wait for the last output from the last logical thread on the previous core (cf. also side note E.2).

Instead, we can modify this program source using the new placement primitives as indicated in listing 11.2. For clarity, we omit the extra logic necessary to handle array sizes that are not a multiple of the number of cores. The top-level function `kernel3` deploys two levels of reduction, where all threads at the first level run on the same core, and each

```
1  double kernel3(size_t n, double *X, double *Z)
2  {
3   sl_place_t pl = sl_default_placement();
4   long ncores = sl_placement_size(pl);
5   long span = n / ncores;
6
7   pl = sl_first_processor_address(pl) | 1 /* force size 1*/;
8
9   sl_create( , /* local placement: */ 1,
10          0, ncores, 1,,,  reductionk3,
11          sl_shfarg(double, Qr, 0.0),
12          sl_glarg(double*, , Z), sl_glarg(double*, , X),
13          sl_glarg(long, , span), sl_glarg(sl_place_t, , pl));
14  sl_sync();
15
16  return sl_geta(Qr);
17 }
18
19 sl_def(reductionk3,, sl_shfparm(double, Q),
20    sl_glparm(double*, Z), sl_glparm(double*, X),
21    sl_glparm(long, span), sl_glparm(sl_place_t, pl))
22 {
23  sl_index(cpuidx);
24
25  long lower = sl_getp(span) * cpuidx;
26  long upper = lower + sl_getp(span);
27
28  sl_create(, sl_getp(pl) + cpuidx * 2,
29       lower, upper, 1,,, innerk3,
30       sl_shfarg(double, Qr, 0.0),
31       sl_glarg(double*, , sl_getp(Z)), sl_glarg(double*, , sl_getp(X)));
32  sl_sync();
33
34  sl_setp(Q, sl_geta(Qr) + sl_getp(Q));
35 }
36 sl_enddef
```

Listing 11.2: Concurrent SL code for the Livermore loop 3, optimized.

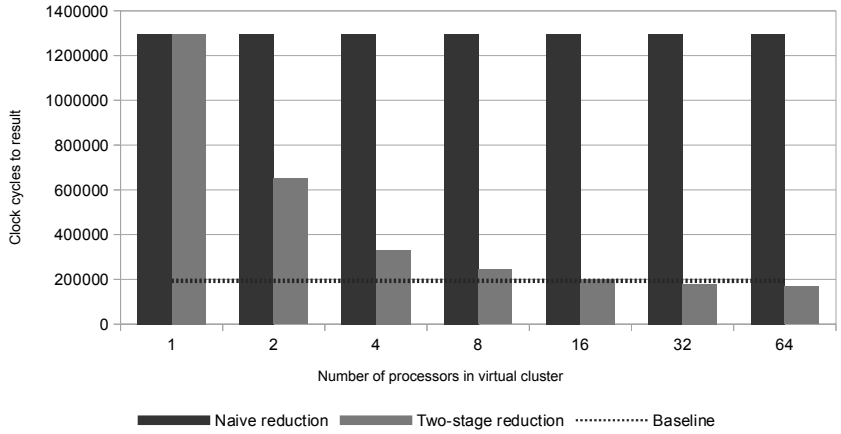


Figure 11.2: Performance of the Livermore loop 3.

family created at the second level runs on a different core within the cluster. The resulting placement at run-time is illustrated in fig. 11.1.

The effect of this strategy on performance is given in fig. 11.2. To obtain these results, we ran the kernel on different virtual cluster sizes, with the input $n = 64000$. As described further in section 13.2 and table 13.1, the microthreaded cores run at 1GHz and the baseline is a sequential program running on 1 core of an Intel P8600 processor at 2.4GHz.

As can be observed, the parallel reduction is advantageous and offers a performance improvement over the baseline past 32 core. As described later in chapter 13, microthreaded cores are expected to be significantly smaller in area than cores in the baseline architecture. The performance per unit of area on chip therefore compares advantageously. Moreover, the proposed design does not use speculation, which suggests that the performance per watt may also be lower. At any rate, multi-core scalability is achieved without *a priori* knowledge about the number of cores while writing the program code.

11.2.5 Relationship with memory consistency

Regardless of how heterogeneous a chip is, and thus how much information on-line resource managers must maintain (cf. section 2.4), a general-purpose, multi-core chip must expose the accessibility of memory from processors as a basic property of the system. Indeed, the *topology of the memory network* impacts the implementation of operating software at all layers of the abstraction stack, as we outlined in section 3.4.1 and chapter 7.

This is mandatory even for so-called "shared memory" chips, because as we explained previously in chapter 7, upcoming and future chip design may feature weak consistency models as an opportunity to simplify and optimize cache coherency protocols. As multiple consistency granularities start to appear, the distributed nature of many-core chips becomes unavoidable to the operating software providers [BPS+09].

To characterize these aspects in a language abstract machine, we introduced the notion of *Implicit Communication Domain (ICD)* and *Consistency Domain (CD)* in section 7.3. These describe subsets of a platform where implicit communication between memory store and load operations is possible between asynchronous threads. Communication is fully

implicit within a CD, whereas it requires some semi-explicit coherency actions in a ICD. Across ICDs, the memory address space is not shared. For example, a Symmetric Multi-Processor (SMP) system with full cache coherency would form a single CD and ICD; different nodes in a distributed cluster would form distinct ICDs; and nodes in a distributed cache system[2] may form a single ICD but distinct CDs.

The placement operators we introduced in the previous sections are tightly related to the memory topology. Indeed, a program code which assumes a single CD cannot safely spread work across a virtual core cluster spanning multiple distinct CDs. For example, on the reference implementation of the proposed architecture, coherency between L1 caches sharing a single L2 cache uses a fully coherent snoopy bus, whereas coherency between L2 caches is lazy: writes may not be propagated to other L2 caches until a thread terminates. This implies that a group of cores sharing the same L2 cache form one CD, but separate L2 caches form separate CDs. A program wishing to use implicit memory-based communication between separate threads, without resorting to explicit memory barriers, must thus ensure that the threads are assigned to cores sharing a single L2 cache. How should this knowledge be negotiated between the hardware implementation and the operating software?

□ As a first step in this direction, we propose the following two primitives:

- `local_consistent_cluster(C)`: for a given cluster address C, produce the address of the largest cluster that encompasses C but provides a single CD. For example, on the proposed architecture this would evaluate to an address for the local L2 core group, whereas on a conventional SMP this would evaluate to an address for the entire local system.
- `local_implicit_cluster(C)`: for a given cluster address C, produce the address of the largest cluster that encompasses C but provides a single ICD. For example, on the proposed architecture this would evaluate to an address for the entire chip, whereas on a distributed chip with separate local memories for each core this would evaluate to an address for the local core only.

We propose that these primitives be used in the run-time system of a higher-level programming language, to select at run-time which implementation to run (whether suitable for a single CD or multiple) depending on the actual resources available locally to a computation.

11.3 On-chip work placement: how to exploit

The primitives provided so far allow a program to control thread placement within a virtual cluster at run-time, including partitioning an existing virtual cluster into physically separate sub-clusters. With the proposed encoding of addressing information, the corresponding operations can be carried out within a few pipeline cycles. This suggests a high level of dynamism, where placement can be decided at a fine grain in algorithms at run-time.

11.3.1 Related work

We have found similar considerations about on-chip placement in the three high-productivity languages for large parallel systems produced during the DARPA project *High Productivity*

[2] For example: `http://msdn.microsoft.com/en-us/magazine/dd942840.aspx`, `http://www.sharedcache.com/cms/`, `http://www.linuxjournal.com/article/7451`

Computing Systems [DGH+08]: Sun's Fortress [ACH+08], Cray's Chapel [CCZ07], and IBM's X10 [CGS+05]. We describe them below.

Also, we supervised an application of our proposal for cross-domain consistency to distributed clusters, which we describe as related work in section 11.3.1.3.

11.3.1.1 Relationship with Chapel and Fortress

Virtual clusters in our setting correspond closely to the concept of *locales* in Chapel and *regions* in Fortress:

> In Chapel, we use the term *locale* to refer to the unit of a parallel architecture that is capable of performing computation and has uniform access to the machine's memory. For example, on a cluster architecture, each node and its associated local memory would be considered a locale. Chapel supports a *locale type* and provides every program with a built-in array of locales to represent the portion of the machine on which the program is executing. [...] Programmers may reshape or partition this array of locales in order to logically represent the locale set as their algorithm prefers. Locales are used for specifying the mapping of Chapel data and computation to the physical machine [...]. [CCZ07, p. 13]

> A *locale* is a portion of the target parallel architecture that has processing and storage capabilities. Chapel implementations should typically define locales for a target architecture such that tasks running within a locale have roughly uniform access to values stored in the locale's local memory and longer latencies for accessing the memories of other locales. As an example, a cluster of multi-core nodes or SMPs would typically define each node to be a locale. In contrast a pure shared memory machine would be defined as a single locale. [Cra11, Sect. 26.1]

> Every thread (either explicit or implicit) and every object in Fortress, and every element of a Fortress array (the physical storage for that array element), has an associated *region*. The Fortress libraries provide a function `region` which returns the region in which a given object resides. Regions abstractly describe the structure of the machine on which a Fortress program is running. They are organized hierarchically to form a tree, the region hierarchy, reflecting in an abstract way the degree of locality which those regions share. The distinguished region Global represents the root of the region hierarchy. The different levels of this tree reflect underlying machine structure, such as execution engines within a CPU, memory shared by a group of cores, or resources distributed across the entire machine. [...] Objects which reside in regions near the leaves of the tree are local entities; those which reside at higher levels of the region tree are logically spread out. [ACH+08, Sect. 21.1]

Like with locale and region variables in Chapel and Fortress, our approach allows programs to manipulate clusters of cores using an abstract handle (a virtual cluster address in our setting). We also suggest in our reference implementation a form of data locality between cores within a locale (cf. fig. 3.9). Furthermore, both Chapel and Fortress provide explicit placement of computations on named locales/regions (with the keyword `on` in Chapel, `at` in Fortress) and relative addressing via the special name "`here`," which corresponds to `sl_default_placement()` in our setting.

Neither Chapel nor Fortress formalize a consistency model for concurrent accesses to memory, like we do. Moreover, our questions about the acquisition of additional resources and how to determine appropriate cluster sizes, mentioned above, do not have an answer in Chapel nor Fortress either.

Despite these similarities, both Chapel and Fortress are considerably more mature with regards to data locality. In particular, both languages define primitives that allow programs to assign variables, especially arrays, to named virtual core clusters. When these constructs are used, any computation using these variables is implicitly distributed across virtual clusters so that each unit of computation becomes local to its operands.

In our setting, this feature is not available natively: variables are either defined in the global scope, in which case they are shared by all clusters visible to the program, or in the local scope, in which case they are only conceptually local to the thread where they are defined and all other threads running on the same core.

In practice, memory locality is obtained automatically by any thread on its local core in our reference implementation, because as seen above this uses a distributed cache protocol which migrates cache lines automatically to the point of last use. However the work on Chapel and Fortress suggests that explicit control of memory placement may be desirable in the language semantics. Moreover, explicit data placement could provide a reliable handle on allocation if our architecture were extended to support multiple distributed memories on chip. We consider this further in section 11.3.1.3.

11.3.1.2 Relationship with X10

Places in IBM's X10 are similar to Chapel's locales and Fortress' regions:

> A *place* is a collection of resident (non-migrating) mutable data objects and the activities that operate on the data. Every X10 activity runs in a place; the activity may obtain a reference to this place by evaluating the constant `here`. The set of places are ordered and the methods `next()` and `prev()` may be used to cycle through them.
>
> X10 0.41 takes the conservative decision that the number of places is fixed at the time an X10 program is launched. Thus there is no construct to create a new place. This is consistent with current programming models, such as MPI, UPC, and OpenMP, that require the number of processes to be specified when an application is launched. We may revisit this design decision in future versions of the language as we gain more experience with adaptive computations which may naturally require a hierarchical, dynamically varying notion of places.
>
> Places are virtual — the mapping of places to physical locations in a NUCC system is performed by a *deployment* step [...] that is separate from the X10 program. Though objects and activities do not migrate across places in an X10 program, an X10 deployment is free to migrate places across physical locations based on affinity and load balance considerations. While an activity executes at the same place throughout its lifetime, it may dynamically spawn activities in remote places [...] [CGS+05, Sect. 3.1]

Like Chapel and Fortress, X10 allows both explicit assignment of computations and data to processing resources, and the local sub-partitioning of resources. With regards to

consistency, X10 defines places as sequentially consistent, which in our model would translate as a constraint that core clusters do not span multiple consistency domains.

The new concept with X10 is the run-time mapping of virtual places (visible by programs) and physical resources (invisible by programs). This reduces the amount of knowledge about the environment provided to programmers but creates the opportunity to optimize load balance at run-time outside of the program specification.

It is conceptually possible to introduce this feature in our setting by indirecting every bulk creation through a system service in charge of placement, which would hold the mapping between place identifiers (manipulated by the program) and physical cluster addresses. The trade-off of this approach is a mandatory overhead at each bulk creation, which would defeat the benefit of architectural support for the effective parallelisation of small workloads. This observation highlights the fact that the primary benefit of hardware support for concurrency, that is low overheads for concurrency creation and synchronization, can only be reaped in applications if the program has direct access to the machine interface.

11.3.1.3 The Hydra run-time

Under our supervision, the author of [Mat10] has thoroughly investigated the use of the proposed consistency semantics across clusters of SMPs. In this work, more SL language primitives have been designed which integrate consistency semantics, placement and object (data) declarations.

This work extends the vision that data should be migrated automatically to the point where it is used, to provide locality implicitly. In contrast to the three "high-productivity" languages mentioned above, the proposed language semantics provide a placement-neutral declaration for variables, and provides instead explicit *declarative* constructs to inform the run-time system *where data may be needed*. The run-time system then exploits this information to determine the minimum amount of necessary communication sufficient to satisfy the consistency semantics assumed in the program. For example, when multiple logical threads explicitly declare inter-domain communication for different subsets of a data item, the run-time system automatically merges these declarations during execution so that only one communication event occurs per node for all logical threads sharing a view on the data item.

11.3.2 Open questions to operating software providers

▷ If the proposed fine-grained interface to placement is to be adopted, operating software providers will need to answer the following questions:

- how does a program obtain access to a virtual cluster initially? Implicitly, we have assumed so far that the top-level thread(s) of a program are placed at some virtual cluster by an "operating system" upon program start-up. With the proposed language constructs, a program can then address cores within this initial cluster. However, system services must be designed to gain access to *additional* clusters if automatic scalability to varying resource availability is desired.
- in an attempt to answer the previous question in our own work, we have prototyped a *cluster allocation service* able to partition the entire chip into sub-clusters on demand. However, we could not find a satisfying *protocol* to query this service: what should be its request parameters?

Our naive initial choice was to define a "desired number of cores" and "whether a single consistency domain is desired" as request parameters. However, this was misdirected: programmers and operating software abstractions do not reason in terms of number of cores and consistency domains; instead, they have real time constraints (e.g. guaranteed minimum throughput on an algorithm) or optimization goals (e.g. maximize the throughput on an algorithm). We are not aware of any existing methodology to translate these requirements to placement requests at a fine grain.

- in more general terms, *can application-level extra-functional requirements on throughput and latency be expressed only as constraints on the number and location of cores involved?* The proposed architecture tries hard to achieve this simplification with its distributed cache, which conceptually provides a uniform view over a shared off-chip backing store and isolates traffic between separate core clusters using a hierarchical topology (cf. section 3.4.1).
  However, we did not find any results that suggest that these properties hold in an actual implementation for larger number of cores. Quite the contrary, using the implementations described in section 4.7 we observed memory bandwidth and latency interference between independent core clusters. We thus cannot exclude that future many-core chips will feature Non-Uniform Memory Access (NUMA) topologies with load-dependent bandwidth and latency properties. Were this to happen, the *communication activity* between program components must be characterized and taken into account by the operating software infrastructure. This would in turn imply that operating software providers will require to know about the semantics of memory consistency protocols on chip, and language designers will need to derive communication requirements between units of work expressed to run concurrently in program source code. We are not aware of any coordinated approach in that direction at the time of this writing.

Nevertheless, the integration of placement at the machine interface creates a *common vocabulary* to reason and talk about these issues between operating software providers. Such a common ground is sorely missing in current architectures, where issues of work placement are resolved using different abstractions in each operating software environment, and are thus difficult to compare analytically. We should thus expect that a common set of primitives will *facilitate* further research activities.

Summary

○■ In this chapter, we have introduced preliminary support for explicit placement of computa-
□ tions to clusters of cores defined dynamically. This support is sufficient to control thread placement in operating software and accelerate multi-stage parallel computations. It also avoids requiring programmers or code generators to make assumptions about the overall topology of the system.

▷ We have also isolated further research questions that must be answered before a full integration of these primitives into higher-level abstractions can be achieved. By studying related work we observe that these questions have not yet been fully answered by other modern, contemporary parallel programming languages; we suggest that common abstractions in the machine interface will facilitate future research in this area.

Chapter 12

Issues of generality

Abstract

In section 1.4.3, we have highlighted that innovation in computer architecture is composed of two parts, one that answers the "inner question" by designing the *substance* of the innovation, and another that answers the "outer question" which places the innovation into its *context* for consideration by external observers. In [Lan07, Lan1x], the author answers the inner question for hardware microthreading in depth, by detailing the hardware design and intrinsically demonstrating its behavior. As part of our process to answer the outer question, we have exposed so far the features of the design visible from software, which we will illustrate in the context of existing software ecosystems in part III. There is however one aspect not covered by the previous chapters of this dissertation nor [Lan07, Lan1x]: the argument to the audience about the *generality* of the invention. In this chapter, we argue that the proposed design provides fully general cores and is suitable to support most parallel programming patterns.

Contents

12.1 Introduction

We consider two levels of generality. The first level is common to any microprocessor design: show whether the proposed design is fit for use for true general-purpose computing, and how the interactive Turing machine is approximated. This reflects the requirements set forth in sections 1.2.1 and 1.3. The second level is specific to a design that offers a parallel execution environment: define what the abstract semantics of communication and synchronization are, how new processes and channels are defined, and what are the conditions under which deadlock freedom and progress are guaranteed.

Usually, in computer architecture, the ancestry of a new component is sufficient to argue for its generality: a new pipeline that supports control flow preemption via interrupts and is connected to a sufficiently large RAM is conceptually sufficiently close to all previous microprocessor designs that its generality can be *implicitly inherited*. Usually, universal and interactive Turing-completeness for individual pipelines stems from conditional branches and the RAM interface to both storage and external I/O, whereas generality relative to parallel execution comes from the availability of time sharing and virtual channels over a consistent shared memory. However, the situation is not so clear if some features are omitted relative to previous approaches, or in the presence of hardware parallelism. For example, the proposed design from part I does away with control flow preemption and consistent shared memory: can it still be used to support general concurrency patterns? It also supports orders of magnitude more concurrent threads than in existing processor designs: what is the cost to guarantee access to a private memory on each individual thread, as required to guarantee Turing-completeness?

An argument for generality can be constructed either by showing that a new design can be described by one or more of the existing theoretical models, or can *simulate* arbitrary programs expressed using these models. Turing-completeness has the Turing machine, λ-calculus or queue machines, as seen in section 1.2.1. For concurrent execution, one can use Hoare's Communicating Sequential Processes (CSP) [Hoa78], Hewitt's Actors [HBS73] or Milner's π-calculus [MPW92a, MPW92b].

In doing so, the finiteness of resources in a concrete implementation should be acknowledged. In general, when some property of an abstract model is dependent on the recursive unfolding of a theoretically unbounded inductive abstract rule, for example the arbitrary scrolling of the tape in a Turing machine, or a property over the replication operator $!P = P|P|P...$ in the π-calculus, the argument about an actual design must show which concrete resource supports the unfolding and which implementation parameter(s) can be tuned to approximate an arbitrarily large specification of a program in the abstract model. In particular, the simulation should be *complexity-preserving*: a basic operation of the simulated model with a constant time and space cost should have a constant maximum time and space cost in the simulation.

As parting words on the topic of the inner question around hardware microthreading, this chapter argues for this generality in the proposed innovation.

12.2 Turing-completeness: one and also many

□ The generality of the design in part I can be directly derived from the generality of *one* individual thread context on chip. As the proposed innovation reuses a standard RISC pipeline, including its support for conditional branches and its memory access instructions, and execution using only thread has access to the entirety of the arbitrarily large external

Side note 12.1: Implementation-independent interleaving.

Our proposed interleaving of N abstract tapes into one is only valid if an upper bound for N is known. As is, our scheme is thus implementation-dependent. However there also exist implementation-independent interleavings. For example, the bits of the context address and the bits of the memory address can be interleaved, as suggested by Morton [Mor66] and Pigeon [Pig01, Sect. 5.3.6.3, p. 115]. The ability to interleave is thus implementation-independent.

memory, the thread can be described by (and simulate) a universal Turing machine, with intuitive cost properties.

An issue arises when considering multiple thread contexts executing simultaneously. For *each* context to be Turing-complete, all must have access to their dedicated, *private* region of memory to simulate their Turing machine's tape. As illustrated in chapter 9, this can be achieved by splitting the logical address space of the entire chip by the actual number of thread contexts on the chip. Other such interleavings can be constructed for any system implementation (cf. side note 12.1). If there are N thread contexts, a Turing machine of any degree of complexity can be implemented by any thread context independently from all others by growing the external memory accordingly by a factor N. This simulation of private memories is further complexity-preserving, assuming the back-end RAM system provides independent, bounded access times to all memory locations.

Finally, we highlight that Turing-completeness is not sufficient *per se*; as mentioned in section 1.2.1, threads should also support interaction, i.e. the ability to interact with the outside world besides RAM. Yet the design from chapter 3 proposes that only some cores are connected to an external I/O interface. For the other cores, the argument for generality can be constructed as follows: as long as an operating software proposes an I/O forwarding service to all cores, where an I/O read or write access from any thread can be delegated, via TMU events over the chip, to a core with physical access to I/O devices, then all cores can be considered general. This simulation of interaction for all cores can further be made complexity-preserving by ensuring that the I/O forwarding services have a fixed overhead, i.e. with a maximum added I/O latency and space requirement for management data structures; this property was e.g. achieved in our implementation of POSIX system services (cf. section 6.4.2); the distributed operating systems Barrelfish [SPB+08] and fos [WA09] also support this pattern.

12.3 Abstract concurrency models

12.3.1 Prior work

○ In [BBG+08, Jes08b, vTJ11, vTJLP09] and other publications, the authors introduce "SVP," a "general concurrency model" derived from hardware microthreading. This model describes the behavior of a program in term of *concurrent families of threads* that are dynamically created and synchronized upon by individual threads. It also defines dataflow channels between threads directly related by family creation, and an asynchronous shared memory where the visibility of writes relative to loads is only guaranteed by family creation and synchronization.

In [VJ07], the authors project the semantics of SVP onto thread algebra [BM07] to demonstrate deadlock freedom under at least the following conditions:

- families are created and synchronized upon by the same parent thread;

- all actions other than reading from a channel are not blocking;
- the dataflow channels are connected so that the directed dataflow graph is acyclic, a thread may only be waiting for a data produced by its predecessors, and all channels are provisioned with a value by their source thread after the source thread has received a value from its predecessor threads.

Furthermore, [VJ07] shows that SVP is deterministic with regards to the values emitted on its dataflow channels and the values written to memory when the program is both deadlock free and race free.

Finally, SVP disallows arbitrary point-to-point communication. Dataflow channels are only connected between a parent thread and the families it directly creates. Communication via memory between concurrent threads is in turn disallowed by stating that writes by one thread are only visible to loads by itself and its successor threads: either those created as new families by the writer thread, or the parent thread after the writer's termination. As such, SVP cannot be used to simulate arbitrary concurrency patterns from CSP, Actors or π-calculus.

This abstract model assumes unbounded resources, in particular that threads can communicate over an arbitrary number of channels and that the creation of a family is a non-blocking operation that always succeeds. As we discussed in chapters 4, 8 and 10, this assumption does not hold for the proposed hardware design, where the maximum number of dataflow synchronizers and bulk synchronizers are bound by physical resources on chip.

To increase the number of channels that can be defined from SVP's perspective, we defined in chapter 8 a method to "escape" channels as slots on the activation records of thread programs. This simulation is also complexity-preserving since each communication operation is simulated by a fixed number of memory addresses and a fixed number of memory stores and loads. However, using this method an arbitrary number of channels can only be created if the thread context performing a family creation has access to an arbitrarily large private memory. For the scheme to work from any context, every context must have access to an arbitrarily large private memory.

Similarly, to increase the number of families that can be created from SVP's perspective, we defined in chapter 10 a method to "escape" creation as a *sequential procedure call*. Again, this simulation is complexity-preserving as each family creation translates to a fixed number of memory addresses and operations. However, using this method an arbitrary number of families can only be created if the first thread context where hardware creation fails supports an arbitrary recursion depth. For the scheme to work from any thread context, an arbitrary sequential recursion depth must be available everywhere. Also, this method only works because the behavior of any family in SVP is serializable into a sequential process.

□ We highlight that the abstract definition of SVP in [VJ07] and other previous publications did not mention explicitly their requirement on arbitrary large private memories and recursion depths from individual SVP threads. As we showed here, these features are mandatory if SVP is to be simulated by a platform with a finite number of concurrency resources. Conversely, if an implementation does not offer arbitrarily large memory and recursion depth per thread, then it does not simulate SVP and is not free of deadlock even when all dataflow channels are provisioned. Of course, if all thread contexts are otherwise Turing-complete, for example as discussed above in section 12.2, both escape mechanisms are implementable and the system can simulate SVP completely.

12.3.2 Communicating sequential processes

We postulate that the proposed hardware design can simulate the semantics of Hoare's CSP in the form described in [Hoa85]. Unfortunately, we have not yet demonstrated this formally; this section merely argues that the platform goes a long way towards supporting CSP, which the SVP model above does not. If the platform can run CSP programs, software audiences can gain confidence that it can also support existing programming abstractions based on CSP such as Occam [MS90], Erlang [AVWW96] or MPI [Mes09].

We consider here only CSP programs that define up to N processes, where N is the number of thread contexts available in the platform. We place this restriction because supporting more than N processes would require interleaving of multiple CSP processes over single thread contexts, which the architecture does not currently support (cf. section 3.3.2). We do not consider this as a strong restriction however: the reference platform we used for evaluation (cf. table 13.1) supports at least 4000+ simultaneously running thread contexts[1]. Further technology advances will likely allow the implementer to increase this limit further.

CSP further exhibits the following features:

- processes can communicate over named channels;
- communication is synchronous, i.e. sending a value in one process and receiving a value in another are a *rendezvous*;
- each process can name (and thus use) an arbitrarily large, but statically known set of channels;
- processes communicate values over channels, and values can be neither channel names nor processes;
- *choice*: processes can wait for an event from two or more channels simultaneously and react as soon as any one channel has an event available;
- channels support communication only in one direction and between two processes [Hoa85, Sect. 4.2, p. 114];
- processes can recurse and activate new processes at every step of the recursion[2];
- concurrent events that do not synchronize processes can proceed concurrently.

Per-process recursion stems from the Turing-completeness of individual thread contexts, discussed previously. The activation of new processes can in turn be implemented using the concurrency management services of the TMU introduced in section 3.3.1. Since separate thread contexts are independently scheduled (cf. section 3.2.1), concurrency of CSP processes mapped to separate thread contexts is guaranteed.

Not considering CSP's support for choice, point-to-point communication can be implemented by the platform's synchronizers and the remote register access primitives of the TMU, described in chapter 3. More specifically, one channel between two processes can be implemented by using two synchronizers at each endpoint, using remote writes to one

[1] It actually supports up to 32000+ contexts if some of them consume less dataflow synchronizers per context than the number allowed by the ISA (cf. section 3.3.3 and chapter 8). However, the number of independent thread contexts is also bound by the number of bulk synchronizers, 32 per core in the reference implementation. Unless multiple CSP processes can be instantiated in a single bulk creation, the number of bulk synchronizers bounds the maximum number of CSP processes that can be created.

[2] In his original paper [Hoa78], Hoare recognized that the semantics of CSP allowed programs to dynamically create new processes but was reluctant to encourage the use of this feature. In the later CSP book [Hoa85], dynamic process creation was explicitly allowed and a note was simply added at the end that some implementations, in particular Occam on the Transputer [MS90] could not create arbitrary numbers of new processes dynamically [Hoa85, Sect. 7.3.6, p. 225].

synchronizer to transmit data and remote writes to the other for acknowledgements. This direct mapping of CSP channels to pairs of hardware synchronizers supports at most $L/2$ channels, where L is the number of private synchronizers allocated for the thread context (cf. chapter 8); it is complexity-preserving since each CSP communication is simulated by a fixed number of synchronizing operations.

Simple pairs of synchronizers cannot implement CSP's choice operation. Choice cannot be implemented over multiple synchronizers, because one thread can wait on at most one synchronizer at a time. Choice from multiple channels cannot be implemented over a single synchronizer either, because concurrent remote register writes to the same target register yield undefined behavior: they are resolved non-deterministically, and the receiving thread does not know that more than one event has been delivered, incurring event loss.

To implement choice, communication must be lifted to *active messages* [vECGS92] instead: sending a message by a CSP thread over a CSP channel must be implemented on our platform by sending a request to create a new thread at the core of the receiving thread. The new thread runs a dedicated "remote delivery" thread program which then stores the value, at the target core, into some incoming buffer and wakes up the thread(s) waiting for input from the CSP channel. This simulation is intuitively complexity-preserving. This basic idea then raises two issues.

The first issue is how to guarantee that there is always a thread context available to receive incoming active messages. This can be obtained by pre-allocating a context on each core prior to program start-up using the TMU's "allocate" event, then storing the context addresses into an array visible from all cores. Then each core willing to communicate would look up the context address from the address of the target core, and send an active message by sending the TMU's "create" event followed by "sync" to the pre-allocated context. "Sync" is required because CSP communication is synchronous: the sender thread must wait until reception is acknowledged by the target mailbox.

The second issue is how to wake up one thread from another on the same core. To implement this, a thread that wishes to wait on one or more channel(s) can reserve a single synchronizer R locally, then set the state of R to *empty*, then write the address of R and the identities of the channel(s) it wishes to wait on in local memory at a location visible from the remote delivery threads on the same core, then start reading from R which causes the thread to suspend. All these primitives can be implemented using the interfaces from chapter 4. When a remote delivery for a given channel arrives at a core, its thread program first reads the synchronizer addresses $R_1, R_2 \ldots$ for that channel from local memory (there may be multiple processes waiting on the same channel), then sends TMU remote register writes of the incoming value to all $R_1, R_2 \ldots$ addresses via the core's loopback NoC interface. Atomicity of access to the memory structures between threads on one core can be negotiated using any of the mechanisms introduced in section 14.1.

The mechanisms introduced above effectively implement arbitrary virtual channels between thread contexts, using one mailbox per core that can be implemented in a local memory. It therefore lifts the limitation on the number of channels introduced above by the direct use of synchronizer pairs. The total number of channels that can be read from in a process is only bound by the size of the local memory at the process' core.

□ To summarize, the proposed platform is likely as powerful as Hoare's CSP, and our proposed simulation of CSP uses only the concurrency control primitives offered by the TMU without requiring a single memory system shared by all cores. It is also universal since the definition of processes and channels can be described fully in software.

12.3.3 Actors

In contrast to Hoare's CSP, the Actor model introduced by Hewitt [HBS73] and developed by Agha [Agh85] is relatively simple:

> *Actors are computational agents which map each incoming communication to a 3-tuple consisting of:*
>
> 1. *a finite set of communications sent to other actors;*
> 2. *a new behavior (which will govern the response to the next communication processed); and,*
> 3. *a finite set of new actors created.*

When an actor receives a message, the system spawns a *task* which computes this 3-tuple. The Actor model does not mandate a specific computational power for the implementation of individual tasks, although [Agh85] suggests that tasks can at least perform arithmetic and simple conditionals.

Communication between actors is further defined to be asynchronous, that is, an actor is immediately ready to accept a new message after sending a message irrespective of whether the target actor(s) have already processed the communicated event. An actor can only communicate with another target actor if it previously knows its *mail address*, i.e. there is no global name space of actors. Each new actor causes the appearance of its own new mailing address, which is known to the creating or created actor, or both.

Following the discussion on CSP above, we can argue that the proposed platform from part I can implement a system of actors. To describe a single actor, an implementation can use a data structure on a local memory on one core which contains a pointer to the task program and its initial parameters, provided when the actor was created. An actor mail address is then composed by pairing the address of the core and the local address of the actor data structure on that core. Then the mailbox mechanism introduced in section 12.3.2 is reused: when a task wishes to send a message to a actor whose address it knows, it sends a "create" TMU event to the target core, providing the local address of the actor's structure as functional argument of the "create" message, and the pointer to a "remote delivery" thread program. The task needs not use "sync" since actor communication is asynchronous. On the target core, the "remote delivery" thread program in turn looks up the actor structure from the provided address in its local memory, then creates (possibly locally) a new thread running the task program specified in the actor structure. It also provides as arguments to the task both the initial actor parameters and the actual payload of the received message. The remote delivery thread can then terminate without waiting on the created task.

To define the "next behavior" of its actor, a task simply overwrites the task and parameters in its own actor's data structure.

Finally, a task that wishes to create a new actor simply requests allocation of an actor data structure from the environment. In [Agh85] the author leaves open *where* the environment should create the actor physically, so an implementation on our platform could choose any core on the system that has some free local memory remaining.

Using these mechanisms, the maximum number of actors that can be active at the same time is bounded by the number of independent thread contexts, e.g. 4000+ in our reference implementation. However, the maximum number of actors that can be *defined*, as well as the maximum number of messages in-flight between actors, is bounded only by the total capacity of the private memory reachable by each core in the system. This can either be

bounded by on-chip memory capacity if cores use local scratchpads, or become arbitrarily large using the techniques from chapter 9. This simulation is complexity-preserving for the behavior of individual actors if the actor tasks are run natively by hardware threads. It can also be made complexity-preserving for communication if a bound is set on the maximum number of actors, so that the look up process at each mailbox has a bounded maximum space and time cost.

□ To summarize, we can construct a simulation of the Actor model on the proposed platform using only TMU primitives and local memories at each core. Our simulation fully exploits the opportunities offered by actors for concurrent execution, both from the behavior independence of distinct actors and the asynchrony of communication. It is furthermore universal since actors can be described fully in software.

12.3.4 π-calculus

Milner's π-calculus is a development from Milner's own Calculus of Communicating Systems (CCS) [Mil80], developed concurrently to Hoare's CSP.

Because CCS and CSP where both inspired from the *zeitgeist* of their times, they share a number of similarities: processes communicate over channels, communication is synchronous, processes can be combined concurrently or via the choice operator, and recursions of concurrent process definitions are allowed. One of the main differences between CCS and CSP is that CCS offers a *restriction* operator, which allows a group of processes to bind a channel name to a private channel not visible by other process groups. In contrast, in CSP the channel name space is shared by all processes.

As in CSP, in CCS channels and processes are in name spaces distinct from the values communicated over channels. The main extension of the π-calculus over CCS was to allow channel names as communicated values. Moreover, the π-calculus supports arbitrary N-to-M communication over single channels.

As in section 12.3.2, we postulate that the proposed platform from part I can simulate the semantics of the π-calculus as described in [MPW92a, MPW92b]. Again, we have not yet demonstrated this formally but we are able to argue so based on simulations.

We start by observing that we cannot directly reuse the simulation infrastructure introduced in section 12.3.2. Although our proposal for CSP can be trivially extended to support a dynamically evolving number of channels, and although the identity of channels could be communicated as values, it binds the receiver endpoint of a channel to a specific core address on chip, where the receiving process is running. This is needed because the active message implementing remote delivery must be addressed to an explicit location on chip. Moreover, so far we considered only one sender and one receiver per channel. In contrast, in the π-calculus, *any* process can send or receive from a channel whose name it knows. Our previous mailbox-based system which assumes that receiving threads are on the same core as the mailbox does not support this.

Instead, we can implement a forwarding service as follows. Any time a process in the π-calculus defines a new channel, its implementation as a thread would send a request at a commonly agreed "channel management" service in the system that would allocate a new channel data structure in the local memory of an arbitrarily selected core in the system to serve as shared mailbox. Any subsequent operation that *sends a channel identity* would then communicate the address of the shared mailbox to the receiver process(es); any time a process *receives a channel identity*, it would receive the address of the corresponding shared

mailbox service and would then inform the shared mailbox that it is now a candidate receiver for that channel. This technique is inspired from the IPv6 "home routing" protocol [PJ96].

Subsequently, any communication *over the channel* would cause two communications, one from the sender thread to the shared mailbox, then from the shared mailbox to the local mailbox of the candidate receiver process(es).

This mechanism indirects communication between two threads implementing π-calculus processes via a third party core, and thus incurs an extra latency compared to the simulations of sections 12.3.2 and 12.3.3. However, each new channel can be instantiated over a different shared mailbox, and the channel management service can thus theoretically spread the communication load over the entire system. This simulation is intuitively complexity-preserving for the sequence of actions by individual processes. As with actors above, it can be made complexity-preserving for communication if a bound is set on the maximum number of channels, so that the look up process at each mailbox has a bounded maximum space and time cost.

□ To summarize, the proposed platform is probably as powerful as Milner's π-calculus. As with CSP above, our proposed simulation uses only the concurrency control primitives offered by the TMU without requiring a shared memory system. It is also universal.

12.4 Turing completeness, take two

In [Mil90], Milner considers *pure* π-calculus, where the only actions that can be performed by processes are the communication events and operators of the π-calculus itself. The author then proceeds to demonstrate that the pure π-calculus is equivalent to Church's λ-calculus, by simulating two evaluation strategies for λ-terms in the π-calculus (one strict and one lazy). His simulations implement λ-terms as processes in the π-calculus, and environment bindings of variables to λ-terms as replicating processes that forward the binding information to other requesting agents implementing β-reductions. Partial β-reductions further execute as concurrent processes in the π-calculus, and thus fully take advantage of the inherent concurrency of the λ-calculus.

With this proof in hand, we can consider a hypothetical implementation of the proposed architecture where some individual thread contexts are not Turing-complete. For example we could reduce the implementation so that some thread contexts have access only to a small amount of local memory, invalidating the techniques from chapter 9. Or we could consider that some thread contexts can use only a smaller instruction set which does not support arbitrary recursion depths. In this heterogeneous platform, we would then be able to distinguish between "fat," general-purpose thread contexts and "light" thread contexts with reduced functionality.

In this setting, the simulation techniques presented in section 12.3.4 can still be used to simulate the π-calculus using "light" contexts. Indeed, the primitives from the section 12.3.4 are implemented in our simulation using only simple synchronizing reads and writes and sending TMU events. The main simulation logic occurs outside of the simulated π-calculus processes, in the mailboxes, where larger memory and recursion are required. However, we can note that only one mailbox per core is necessary; an arbitrary number of "light" thread contexts can be used around each mailbox without changing the semantics of the simulation.

□ In such an environment with a few "fat" mailbox thread contexts and many "light" thread contexts, the system would be able to carry out the reduction of arbitrary λ-terms as per [Mil90], by spreading the evaluation over a dynamically defined, arbitrarily large network of small communicating processes. In other words, the system *as a whole* would be

Turing-complete. Moreover, its expressivity power could expand arbitrarily by increasing the number of "light" thread contexts, at a fixed "fat" context budget. The system would also be universal because processes and channels would still be specified by software. The only drawback of reducing the number of "fat" contexts is that they are a bottleneck for communication: a smaller number of "fat" contexts implies less available bandwidth overall.

Unfortunately, we were not able to determine yet whether this simulation would be complexity-preserving, that is whether a bound can be set on the space and time costs to simulate one execution step of the simulated Turing machine.

12.5 Exposing the generality to software

The arguments above merely *suggest* that the design is general. There are two ways forward to convince audiences more thoroughly. The first is to describe the semantics of the hardware interface from chapter 4 precisely in an abstract model, and then prove formally that this model is at least as expressive as other existing models. This approach, while necessary to gain credibility in theoretical circles, would be however largely inefficient in making the invention accessible. Instead, more popular audiences will require *programming languages* and *frameworks* that expose the system's potential to their creativity (cf. section 1.3).

So far, our contributions provide a general-purpose programming environment for individual cores (the C language), and a handful of concurrency-related features:

- constructs to spread work over multiple thread contexts and cores, with semantics related to the SVP model described in section 12.3.1;
- a memory consistency model where updates by unrelated concurrent threads may be visible to each other (namely, if they are part of the same CD, cf. chapter 7);
- the foundations of a finite resource model (chapters 10 and 11).

▷ These features makes our contribution only barely more powerful than the SVP model described in section 12.3.1. In particular, to implement the CSP, Actors and π-calculus simulations introduced in section 12.3, the language interface must be further extended with operators to:

- take the address of a synchronizer;
- issue a remote access to a synchronizer whose address is known;
- reserve a thread context ("allocate") and store its identity;
- create threads in a previously reserved context[3];
- synchronize on termination of threads without releasing the context[4];
- organize atomicity of access to memory between threads running on a single core.

We estimate that little effort is required to add these primitives to the framework introduced in chapter 6, since these primitives are already available in the hardware interface from chapter 4. However, we also highlight that any program that would subsequently use these features would not necessarily be serializable any more, limiting the user's ability to troubleshoot programming errors (cf. section 6.2.4). This may warrant the separation of these additional services in either a separate *system language*, or a set of *privileged* language constructs that would be only usable by the operating software of higher-level parallel languages, and not directly by application programmers. Alternatively, an emulation of these

[3] Currently, reservation and creation are bound in a single creation construct.

[4] Currently, synchronization and release are bound in a single synchronization construct.

primitives could be implemented on a legacy platform to serve as reference for troubleshooting. We suggest that this exploration in language design be performed in future work.

12.6 Generality in other designs

Most contemporary SMP chip designs inherit their generality in an almost boringly simple way. For example, each individual core in Intel's general-purpose multi-core offerings, including the P6, NetBurst, Core, Nehalem, Atom, Sandy Bridge micro-architectures, combine traditional general-purpose ISAs with a timer interrupt, coherent views on a common shared memory and direct access to system I/O; so do AMD's K8-K10 and Sun's/Oracle's SPARC multi-core products. The timer interrupt in turn allows programmers to define arbitrarily large numbers of virtual processes interleaved using a software scheduler; the coherent shared memory enables arbitrarily large numbers of virtual channels connected in arbitrary patterns. Universality and interactivity on each core are evident, as every behavior is defined in software and all cores have symmetric access to I/O. We can find more diversity in other "exotic" designs that have surfaced in the last 10 years.

In IBM's Cell Broadband Engine [KDH+05], a general-purpose PowerPC core called PPE is combined with 8 to 16 smaller RISC cores called SPEs. Although each SPE supports a general instruction set featuring conditional branches, and direct access to the outside world via its own Direct Memory Access (DMA) controller, it is connected only to a local RAM with a capacity of 256KiB which contains both code and data. This capacity may be sufficient to accommodate the workload of sub-computations driven from the PPE; however it seems to us insufficient for general-purpose workloads driven directly from the SPE. Meanwhile, each SPE supports inter-SPE communication via 128 distinct synchronous channels implemented in hardware. This makes the SPE group a suitable platform for simple process networks with up to 8-16 processes (the number of SPEs). Although each SPE supports control flow preemption via external interrupts, this feature could not simply be used by a software scheduler to virtualize more processes because the channel read and write operations are not interruptible.

In NVIDIA's Fermi [LNOM08, NVI09] GPGPU designs, each core, called an SM, is equipped with a threading engine able to schedule multiple independent threads concurrently. Each thread can run arbitrarily patterns of conditional branches and can be configured to access an arbitrarily large private memory via a unified cache to external RAM (a feature not present in NVIDIA's previous GPGPU offerings). As such, each thread features Turing-completeness. However, Fermi threads lack the generality required in section 1.2.1: they cannot be interactive, because Fermi does not allow GPGPU threads to access external I/O devices.

In terms of concurrency patterns, Fermi threads do not support control flow preemption via a timer interrupt and thus cannot multiplex multiple logical processes over one thread context. However, Fermi does support inter-thread synchronization within one SM using atomic accesses to a local, 16KiB shared memory. Since each SM can run 768 separate threads, the chip should thus support CSP and π-calculus up to that number of processes within each SM, using mechanisms similar to those we propose later in sections 12.3.2 and 12.3.4, with the number of channels limited by the local memory capacity. When considering the entire chip of multiple SMs instead, communication between SMs is possible via the external RAM, however memory atomics do not cross SM boundaries so synchronization would require busy waiting. To summarize, Fermi's architecture can support general

concurrency patterns efficiently within one SM, and less efficiently across all SMs in one chip.

We also considered Intel's Single-Chip Cloud Computer (SCC) [MRL+10], a research platform, and Tilera's TILE64 [BEA+08], a product offering for network applications. Both integrate a larger number of general-purpose cores on one chip than contemporary multi-core product offerings: 48 for the SCC, 64 for TILE64. All cores are connected to a common NoC. On both chips, Turing-completeness at each core is achieved by a traditional design—the MIPS pipeline for TILE64, the P54C pipeline for the SCC—and a configurable mapping from cores to external memory able to provide the illusion of arbitrarily large private memory to each core. Interactivity is provided on the TILE64 by direct access to external I/O devices on each core via dedicated I/O links on the NoC; on the SCC, the NoC is connected to an external *service processor* implemented on FPGA which forwards I/O requests from the SCC cores to a host system. The SCC approach to I/O is thus similar to the one we took in chapter 5.

For parallel execution, TILE64 and SCC only support one hardware thread per core, but cores feature preemption as the means to multiplex multiple software processes. TILE64 offers comprehensive support for communication: it supports 4 hardware-supported asynchronous channels to any other cores (UDN), a single channel to a configurable, static set of peers (STN) and two dedicated channels to external I/O devices per core (IDN). Alternatively, cores can also implement virtual channels over a coherent, virtually shared memory implemented over another set of NoC links (MDN), although the communication latency is then higher. This diversity of communication means ensures that the design can support most general parallel programming patterns.

In contrast, the SCC does not offer a coherent view of shared memory to cores. While each pair of cores has access to a local scratchpad of 256KiB, called Message-Passing Buffer (MPB), which can be accessed remotely by other cores via the NoC, the MPB does not synchronize. Instead, point-to-point synchronization can be negotiated only via IPIs, or by disabling the local caches and busy waiting on changes to external memory regions. As such, while the SCC theoretically supports most general parallel programming patterns, its actual implementation yields poor point-to-point communication latencies.

Summary

When designing new components as building blocks for computing systems, the innovator should describe the level of semantic generality provided by the invention. Especially when designing components for *general-purpose* systems, generality should be argued by relating the new component to theoretical models whose generality has been previously established. In most contemporary microprocessor designs, generality is implicitly inherited by reusing the traditional model of Turing-complete processors implementing timer-driven control flow preemption and connected to a shared memory that can implement arbitrary point-to-point communication. These basic conceptual models inherit Turing completeness from the individual processors and the semantics of most abstract concurrency models developed since the 1970's.

In contrast, the proposed CMP design that we covered earlier does not provide preemption. It supports but does not require a shared memory. Also, it provides a large number of thread contexts which compete for a single address space. Because of these differences, a new bridge must be constructed between this design and existing abstract models before it can be advertised as "general."

○ In this chapter, we have acknowledged previous work in this direction by our peers, where a "concurrency model" named SVP was defined. We also showed the limitations of this approach.

□ Instead, by constructing simulations using only the platform's dedicated hardware concurrency management primitives and private memory on each core (i.e. without requiring a shared memory system), we were able to relate it to Hoare's CSP, Hewitt and Agha's Actors and Milner's π-calculus. Furthermore, using our contribution from chapter 9, we were able to regain Turing-completeness for individual cores under the assumption of arbitrary large external memory. If this latter assumption does not hold, we are still able to suggest

▷ Turing-completeness for the entire system based on support for the π-calculus. Our simulations are based on the machine primitives described in chapter 4. These primitives are not all yet exposed in the language interface from chapter 6; therefore, future work must provide further language support before the full generality of the platform becomes available to external audiences.

Part III

Applications and experiences

—Feedback on the answers to the inner and outer questions

Chapter 13

Core evaluation
—Lessons learned about hardware microthreading

Abstract

Once a platform is defined around an architectural innovation, effort must be invested into engaging with the audience and gaining feedback about the innovation. In this chapter, we provide an example of this interaction. In our ecosystem, a comprehensive evaluation of the architecture and its implementations was realized. We highlight some key results from this evaluation, for two purposes *besides* the evaluation results themselves. One purpose is to document how the platform definition from part II is practically related to the actual evaluation work, i.e. what interactions actually took place. The other is to illustrate that the interaction with our audience has enabled early feedback on the architecture design, as suggested in section 5.3.

Contents

13.1 Introduction

As we explained in section 5.4.2, we worked in an ecosystem specifically set up to evaluate the innovation from part I. We faced three direct audiences: a research partner organization in charge of demonstrating the benefits of SAC [GS06] to program the proposed architecture, a partner organization in charge of discovering fine-grained concurrency in plain C loops, and various individuals in charge of hand-writing benchmarks using C and our proposed language extensions. In section 13.2 below, we explain through a running example the methodology used by our audience to carry out the platform evaluation. We also explain how our contributions from part II were useful in this context.

While we participated in these activities merely as support staff, we gained insight into the architecture. As this insight has not yet been published elsewhere, we share it in the remainder of this chapter. In particular:

- in section 13.3, we explain how we can separate the impact of multithreading from the impact of multi-core execution to understand performance results;
- in section 13.4, we identify dynamically heterogeneous, functional concurrency and introduce an architectural feature that we co-designed to optimize this use case;
- in section 13.5, we illustrate issues of power usage;
- in section 13.6, we illustrate throughput-oriented workloads;
- in section 13.7, we identify high-level issues shared by other multi-core designs;
- in section 13.8, we identify that a hallmark feature of the proposed design, the sharing of synchronizers by adjacent thread contexts, is actually of limited use and could be advantageously removed in favor of other, simpler features.

13.2 Evaluation efforts from the ecosystem

The benchmarking activities were organized around a single common theme: produce speedup diagrams that demonstrate the scalability of a single code representation across different hardware configurations. To achieve this, all the benchmarks have been written to share a common pattern:

1. load input data in memory;
2. sample performance counters;
3. execute a workload;
4. sample performance counters, and report the differences;
5. optionally, execute steps 2 to 4 multiple times.

Each program is then:

1. compiled once using our tools from chapters 6 and 8,
2. run multiple times on our platform from chapter 5, by placing it at start-up on core clusters of different sizes (from 1 to 64 cores) using the primitives from chapter 11.

Most programs further required the features detailed in chapters 9 and 10. The outcome for each benchmark is a series of samples which report the time to completion, the number of instructions executed, and other variables relevant to the evaluation of a processor architecture.

□ In addition to the features from part II, we implemented the following additional operating software for the benchmarking activities:

Side note 13.1: About the relevance of the Livermore loops.

We acknowledge that the Livermore kernels are extremely small kernels unrepresentative of contemporary large applications. They were designed to be representative of loops in large high-performance applications and were selected/designed to test how effective vectorising compilers were at recognising concurrency in these applications. This benchmark suite is nowadays mostly superseded by newer developments that also test workloads from outside the HPC community, e.g. the more diversified Standard Performance Evaluation Corporation (SPEC) and NASA Advanced Supercomputing (NAS) benchmark suites. We discuss this further in chapter 15.

```
1  COMMON /SPACE1/ U(1001), X(1001), Y(1001), Z(1001)
2  COMMON /SPACER/ Q, R, T
3  ...
4  cdir$ ivdep
5   1007 DO 7 k= 1,n
6           X(k)=     U(k   ) + R*( Z(k   ) + R*Y(k   )) +
7       1         T*( U(k+3) + R*( U(k+2) + R*U(k+1)) +
8       2         T*( U(k+6) + Q*( U(k+5) + Q*U(k+4))))
9      7 CONTINUE
```

Listing 13.1: FORTRAN code for the Livermore loop 7.

- a resource management service that reserves a cluster of cores upon system start-up and starts the benchmark program's `main` function on this cluster;
- a performance counter sampling framework providing a uniform API to programs across all target implementations;
- a custom data input API able to load large arrays of data from files.

Beyond measuring performance over multiple core cluster sizes, each benchmark was also used to experiment with different compiler optimization flags, both at the higher-level SAC or parallelizing C compiler, and the SL tool chain. Each benchmark was also used to experiment with different architecture parameters, e.g. number of cores, cache sizes, etc. These benchmarking activities were spread over the ecosystem; an exhaustive report of all the benchmark results would be outside of our scope. Instead, we focus below on one benchmark to illustrate how the work was carried out in our technical framework.

13.2.1 Running example: Livermore loops

The Livermore FORTRAN kernels [McM86] are a sequence of 24 algorithms taken from scientific code. Their reference implementation is a FORTRAN program that exercises all 24 algorithms multiple times and computes a statistical report of their performance.

13.2.2 Implementation

Each of the 24 kernels was extracted individually from the FORTRAN implementation. One partner rewrote the kernels using SAC; separately, using a translation of the FORTRAN code to C as a basis, one partner used their parallelizing C compiler to automatically discover concurrency in the sequential C code and replace loops by uses of our language extensions, whereas we did the same work manually. We depict this implementation work in fig. 13.1.

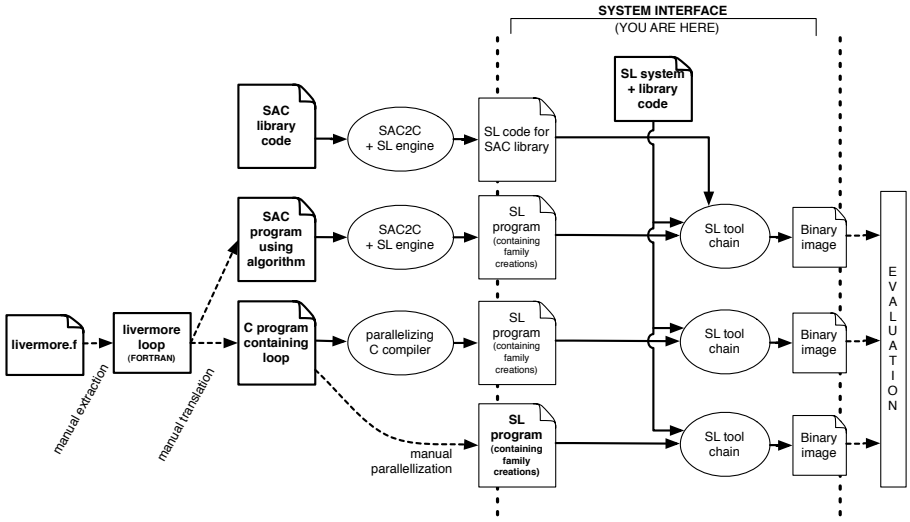


Figure 13.1: Implementing the Livermore loop benchmarks using our proposed framework.

```
1  double u[1001], x[1001], y[1001], z[1001];
2  double q, r, t;
3  ...
4  for ( k=0 ; k<n ; k++ ) {
5    x[k] = u[k] + r*( z[k] + r*y[k] ) +
6      t*( u[k+3] + r*( u[k+2] + r*u[k+1] ) ) +
7      t*( u[k+6] + q*( u[k+5] + q*u[k+4] ) ) );
8  }
```

Listing 13.2: Sequential C code for the Livermore loop 7.

```
1  specialize
2  double[+] Loop7( int n, double q, double r,
3          double t, double[1001] u, double[1001] y,
4          double[1001] z);
5  double[+] Loop7( int n, double q, double r,
6          double t, double[+] u, double[+] y,
7          double[+] z)
8  {
9    a = u + r * (z + r * y)
10     + t*(shift([-3], u) + r*(shift([-2], u) + r*shift([-1], u))
11     + t*(shift([-6], u) + q*(shift([-5], u) + q*shift([-4], u))));
12   return (take([n], a), inter);
13 }
14 ...
15 x = Loop7( n, q, r, t, u, y, z );
```

Listing 13.3: SAC code for the Livermore loop 7.

```
1  sl_def(innerk7,, sl_glparm(double*, x),
2    sl_glparm(double*, u), sl_glparm(double*, z),
3    sl_glparm(double*, y), sl_glfparm(double, r),
4    sl_glfparm(double, t), sl_glfparm(double, q))
5  {
6    sl_index(k);
7    double *x = sl_getp(x), *u = sl_getp(u),
8         *z = sl_getp(z), *y = sl_getp(y),
9         r = sl_getp(r), t = sl_getp(t), q = sl_getp(q);
10
11   x[k] = u[k] + r*( z[k] + r*y[k] ) +
12                 t*( u[k+3] + r*( u[k+2] + r*u[k+1] ) +
13                 t*( u[k+6] + q*( u[k+5] + q*u[k+4] ) ) );
14 }
15 sl_enddef
16 ...
17 double U[1001], X[1001], Y[1001], Z[1001];
18 double Q, R, T;
19 ...
20 sl_create(,,, n,,,, innerk7,
21   sl_glarg(double*,, X), sl_glarg(double*,, U),
22   sl_glarg(double*,, Z), sl_glarg(double*,, Y),
23   sl_glfarg(double,, R), sl_glfarg(double, T),
24   sl_glfarg(double,, Q));
25 sl_sync();
```

Listing 13.4: Concurrent SL code for the Livermore loop 7.

To illustrate further, we focus on one particular kernel, the equation of state fragment, whose original FORTRAN code is given in listing 13.1. This was separately translated to an equivalent C loop (listing 13.2) and parallel SAC code (listing 13.3); we then manually encapsulated the C loop body in a thread program and to produce the concurrent SL version in listing 13.4. Compared to the C code, we explicitly lift the reference to the globally declared variables as thread program channels in our SL code to avoid an external symbol reference in every thread, because our implementation strategy prevents the underlying C compiler from automatically detecting common sub-expressions across thread functions. Otherwise, no difficulty is introduced: the sequential loop of n iterations is replaced by a family creation of n logical threads. Another Livermore loop example using semi-explicit work placement to perform parallel reductions was also provided in chapter 11.

13.2.3 Results

Some example results for this benchmark are illustrated in figs. 13.2 to 13.4. Both the cycle-accurate, many-core, microthreaded platform emulation and a legacy architecture were used for comparison. The system characteristics of the legacy platform and the microthreaded platform for the results of this chapter are listed in table 13.1.

The code was compiled once (fig. 13.1) using version 3.6b of the SL tool chain, relying on GCC version 4.5 as an underlying code generator for the microthreaded architecture and GCC 4.2.1 to compile the sequential C code to the legacy architecture. The default compiler settings were used for optimization ("`-O2`").

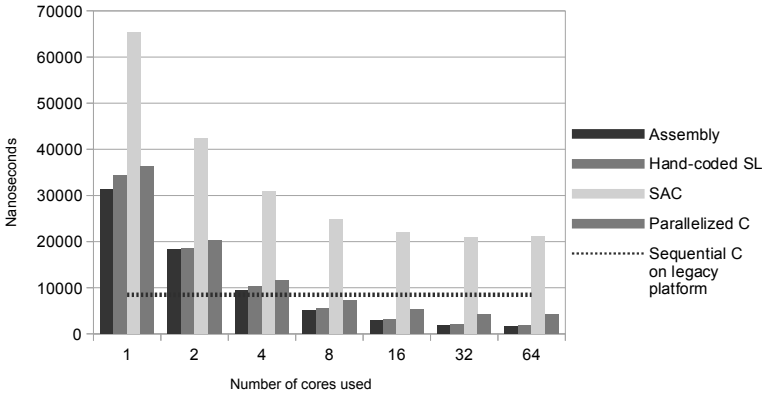


Figure 13.2: Time to result (performance) for the Livermore loop 7.

Problem size and baseline are described in section 13.2.3.

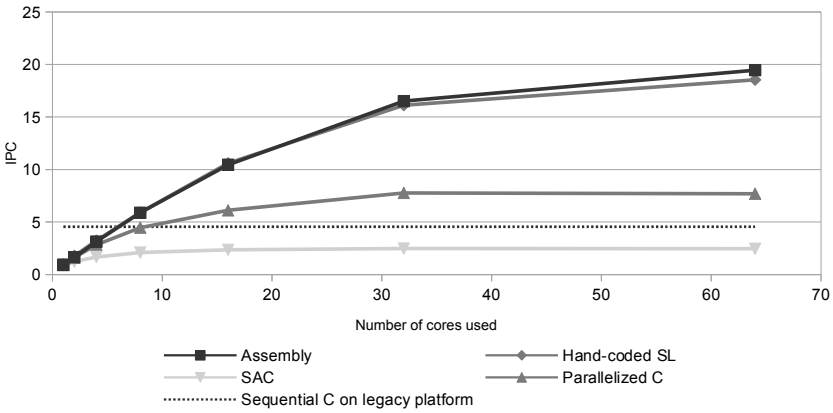


Figure 13.3: Instructions per cycle (utilization) for the Livermore loop 7.

Problem size and baseline are described in section 13.2.3.

For this benchmark, the input $n = 990$ was used as per the original Livermore benchmark specification. We used two baselines for comparison: the first is the behavior of the sequential C code on the legacy system, and the second is a hand-written, hand-tuned raw assembly program for the microthreaded platform written by an architecture expert.

There are different types of observations to draw from these results, depending on the audience.

13.2.3.1 Observations from the architect's perspective

The results have illustrated that this specific implementation is able to scale performance for small workloads, e.g. 990 microthreads in the results above (each performing one iteration of the original loop) over multiple cores, up to dozens of cores, e.g. 32 cores above. This is remarkable as such small workloads (less than 10^5 instructions) would not be able to compensate concurrency management overheads (10^6 instructions or more) on a legacy

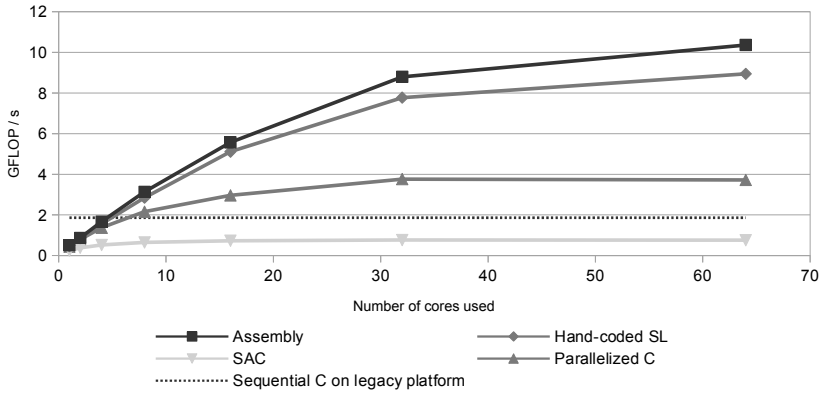


Figure 13.4: Floating-point performance for the Livermore loop 7.

Problem size and baseline are described in section 13.2.3.

	Legacy platform	Microthreaded platform
System	MacBookPro7,1	MGSim v3
Processor chip	Intel Core 2 Duo P8600	Many-core MT chip
Core micro-architecture	Intel Penryn	In-order 6-stage RISC pipeline + microthreading
Issue width	4	1
ISA	x86-64	DEC/Alpha (64-bit) + MT extensions
Core frequency	2.4GHz	1GHz
Number of cores	2	128
FPUs	2	64
Hw. threads	2	32640
Hw. threads / core	1	255
L1 cache (total)	128K	768K
L2 cache (total)	3MB	4MB
L1 cache / core	64K	6K
Memory interface	1x DDR3-1066	4x DDR3-1600†
RAM in system	4GB	4GB
Chip technology	45nm	45nm (est.)
Chip area	107mm^2	120mm^2 (est.)

† The implementation for the DDR3 interface was incomplete, see section 13.7.1 for details.

Table 13.1: System characteristics.

architecture. This result, repeated over most other benchmarks, confirms that the low-overhead hardware support for concurrency management in the proposed design is able to exploit more software concurrency than traditional software-based approaches.

For larger numbers of cores, the performance then saturates. Various effects cause this limitation, including memory access latencies, overhead of communicating concurrency management events over a larger number of cores, memory bandwidth, load imbalance. Again, a full analysis of the performance characteristics are outside of the scope of our dissertation.

Nevertheless, we were particularly excited to find that the performance of the 1-core legacy baseline was matched in multiple benchmarks by less than 32 microthreaded cores, e.g. 8 cores in the running example above. Considering the technology details from table 13.1, this corresponds to a smaller area on chip than 1 core of the selected legacy platform, which was close to the state of the art at the time of this writing. In other words, for this specific benchmark the performance per unit of area is higher with the new architecture. Combined with the observation that the proposed design does not use speculation in any way, contrary to the legacy design, these results suggest that the performance per watt is also higher.

13.2.3.2 Observations from the software engineer's perspective

The naive rewrite of the sequential C loops (e.g. listing 13.2) as a thread family (e.g. listing 13.4) is sufficient to obtain, after compilation through our tool chain, performance figures close to the hand-optimized assembly code (e.g. more than 85% in the example). The overhead of using SL comes mainly from the absence of global common sub-expression elimitation (across thread programs), which forces array base address to be recomputed in every logical thread. In other words, despite our simple and coarse approach to compilation in chapter 6, we could successfully rely on the interface language to expose the performance opportunities of the new architecture.

The preliminary results as to the applicability of the new design to existing software code bases are also encouraging. Indeed, the benchmarks show that code automatically parallelised from C using the partner technology, reported on in [SEM09, SM11] can attain both a higher performance than the legacy baseline within the same chip area budget, and also multi-core performance scalability, cf. e.g. fig. 13.2.

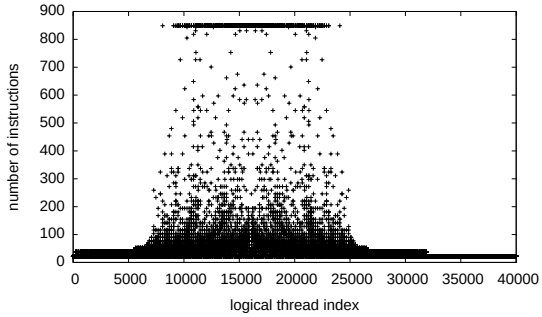
13.3 Separate impact of multi-threading and load balancing

The evaluation activities have revealed that the key parameters to optimize execution efficiency are multi-threaded execution per core and techniques to optimize load balance across cores, and these are mostly orthogonal.

To illustrate this, we use an example program which exposes a heterogeneous workload. The workload is defined by a bulk creation of 40000 logical threads where the amount of work per thread varies irregularly, as illustrated in fig. 13.5.

We show the performance of this workload in the same environment as section 13.2 (table 13.1) in fig. 13.6. We used two implementations that differ in how the logical threads are distributed across cores:

- in the "even" implementation, the logical range is divided into P equal segments where P is the number of cores in the cluster, i.e. core p runs indices $\{start_p, start_p{+}1, start_p{+}2 \ldots\}$. This is the straightforward use of the hardware bulk creation process, by requesting a single bulk creation over all cores in the cluster;



Insns. per thread	Number of logical threads	Instructions accounted
0–83	35524 (88.81%)	1141009 (31.45%)
84–167	1314 (3.29%)	145456 (4.01%)
168–251	332 (0.83%)	65760 (1.81%)
252–335	120 (0.30%)	34824 (0.96%)
336–419	64 (0.16%)	23518 (0.65%)
420–503	36 (0.09%)	16448 (0.45%)
504–587	32 (0.08%)	17596 (0.49%)
588–671	16 (0.04%)	9968 (0.27%)
672–755	14 (0.04%)	10126 (0.28%)
756–839	14 (0.04%)	11504 (0.32%)
840–923	2534 (6.33%)	2151336 (59.31%)
(40000 logical threads, 3627545 instructions total)		

Figure 13.5: Actual thread sizes in the example heterogeneous workload.

Side note 13.2: Description of the example heterogeneous workload.

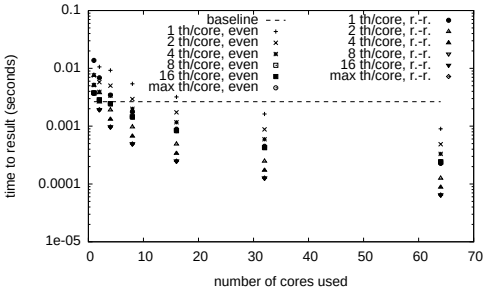
Each logical thread l in the range $[0, 40000]$ evaluates the function $m(l)$ defined by

$$\begin{cases} m(l) = f\big(x_{start} + x_{step} \times (l \mod N) + i\,(y_{start} + y_{step} \times \lfloor l/N \rfloor)\big) \\ f(c) = min\,(\{k \mid 2 \leq |z_k|\} \cup \{64\}) \quad \text{with } z_0 = c \text{ and } z_{n+1} = z_n^2 + c \\ x_{step} = (x_{end} - x_{start})/N \\ y_{step} = (y_{end} - y_{start})/M \end{cases}$$

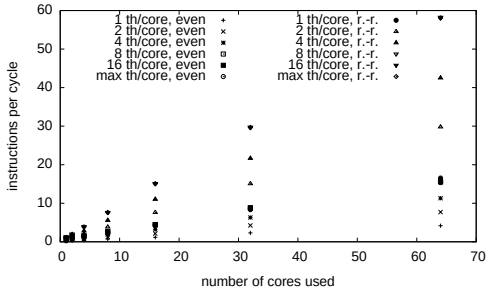
where $(x_{start}, x_{end}, y_{start}, y_{end})$ are input parameters that define a window over the complex plane, and (N, M) are input parameters that define the discretization of this window. We use:

- $N = M = 200$, hence $N \times M = 40000$ logical threads;
- $x_{start} = y_{start} = -2$, $x_{end} = y_{end} = 3$.

The function f computes which iteration of the complex quadratic polynomial $z_{n+1} = z_n^2 + c$ first escapes the closed disk of radius 2 around the origin, with a maximum of 64 iterations. This is the function typically used to visualize the boundary of the Mandelbrot set [PR86]. We provide the corresponding source code in Appendix K.

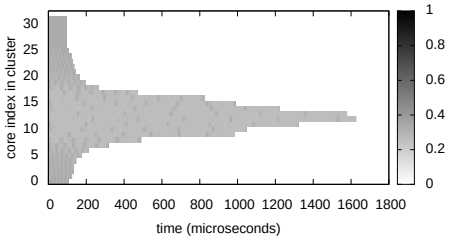


(a) Time to result.

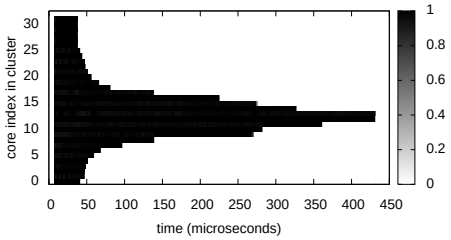


(b) Instructions per cycle.

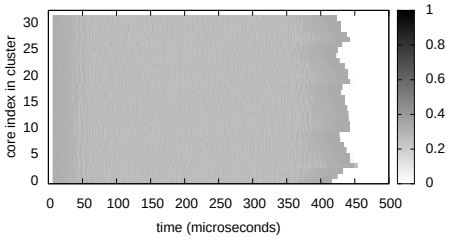
Figure 13.6: Performance of the example heterogeneous workload.



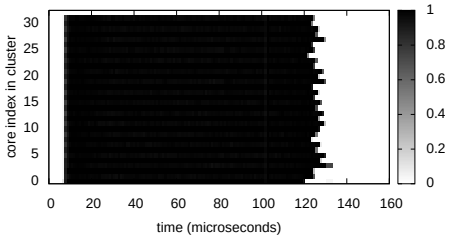
(a) Even distribution, 1 thread/core.



(b) Even distribution, 16 threads/core.



(c) Round-robin distribution, 1 thread/core.



(d) Round-robin distribution, 16 threads/core.

Figure 13.7: Per-core activity for the example heterogeneous workload running on 32 cores.

- in the "round-robin" implementation, the logical range is distributed in a round-robin fashion over the P cores in the cluster, i.e. core p runs indexes $\{p, p+P, p+2P \ldots\}$. This triggers bulk creation separately with different logical index ranges on every core of the cluster.

As can be observed in fig. 13.6, the performance of the proposed platform exceeds the reference baseline consistently past 32 cores, and for some parameters beyond 2 cores. This is compatible with the observations from section 13.2.3.1.

Furthermore, fig. 13.6 reveals that the round-robin distribution is radically beneficial to performance. The reason why this is so is exposed more clearly in fig. 13.7: with the even index distribution, the heterogeneity of the workload causes imbalance between cores, whereas the round-robin distribution exploits the local homogeneity of the computational problem to spread the workload more evenly across cores.

This example illustrates the following:

- per-core multithreading is effective at increasing per-core utilization, i.e. instructions per cycle on each core, regardless of load distribution. This can be observed in fig. 13.7b relative to fig. 13.7a and fig. 13.7d relative to fig. 13.7c. While this is a well-known effect for I/O- or memory-bound workloads, this benchmark confirms that fine-grained multithreading is also effective at tolerating latencies of FPU operations, which are handled asynchronously in the proposed architecture.
- the default logical index distribution performed by the hardware bulk creation process, primarily designed for the deployment of regular data-parallelism across cores, i.e. as

Side note 13.3: Choice of QuickSort for evaluation.

QuickSort was chosen as an instance of *general*, *unstructured* divide-and-conquer algorithm, to evaluate load balancing when no application-specific knowledge is known. In particular no consideration was given as to whether it was the fastest algorithm to sort arrays of integers. We acknowledge that for this specific application, using instead an algorithm based on sorting networks [Bat68],[Knu98, pp. 219–247],[PF05, Part 6] would make most efficient use the parallelism available on the target architecture.

a general-purpose substitute of specialized SIMD units, cannot be naively applied to heterogeneous workloads to *maximize* performance. For example, figs. 13.6 and 13.7 reveal that the even distribution yields 3× to 4× higher execution times than the round-robin distribution for this workload.

- Nevertheless, even with load imbalance and sub-optimal performance, the higher performance density of the proposed design may allow a naive implementation to exceed the performance of a legacy design in the same area budget, as demonstrated in fig. 13.6 from 8 cores.
- As shown in this example, maximum performance can be approximated with a round-robin distribution when the workload is known to be locally homogeneous despite the overall heterogeneity. This application-specific decomposition domain transposition is further facilitated by the full configurability of the $(start, limit, step)$ parameters to the hardware creation process.

▷ This latter observation suggests capturing the distribution of globally heterogeneous, but locally homogeneous workloads in a programming language construct at some level of abstraction. The exploitation of this opportunity in higher-level programming languages should thus constitute future work.

The next section explores cases where the application domain does not provides good static decompositions.

13.4 Applicability to irregular functional concurrency

There is limited support in the platform to resolve load imbalance in functionally concurrent programs where the amount of work per sub-problem is dependent on the input. An example was provided by the evaluation of QuickSort using SAC on the proposed platform (cf. side note 13.3). QuickSort uses divide-and-conquer concurrency where the depth of any sub-tree, and thus the complexity of any spatial sub-part of the concurrent workload, is dependent on the values to sort. The design from part I does not provide any hardware support for dynamic load balancing; instead, the QuickSort evaluation carried out by our audience used different explicit placement strategies using our proposed operators from table 11.1.

An example performance graph is given in fig. 13.8; it reports the time to sort 1024 integers using three implementations: one using the current core and the next in the cluster at each divide step (fig. 13.9a); another using the previous and next cores (fig. 13.9b); the last using the upper and lower half of the current cluster at each divide step (fig. 13.9c). Although fig. 13.8 shows relatively good scalability of the last implementation up to 16 cores, the work distribution is still strongly imbalanced, as shown in fig. 13.9d.

■ In an attempt to minimize such dynamic load imbalances, we jointly co-designed an extension to the hardware bulk creation context with our audience, where the “allocate” message sent to a cluster (cf. section 3.4.2) would travel two times through all cores, one time to select the least used core and the second to actually reserve the thread context.

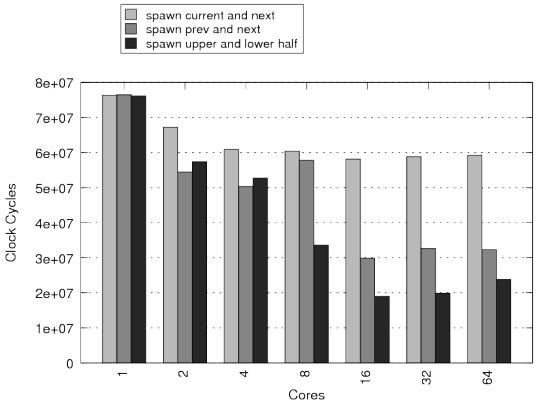
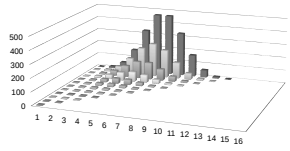
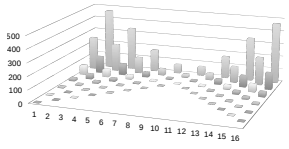


Figure 13.8: QuickSort performance on the proposed platform.

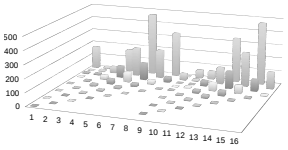
Figure reproduced with permission from [SHJ11].



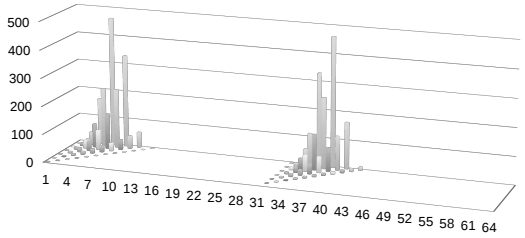
(a) Current/next distribution.



(b) Previous/next distribution.



(c) Lower half/Upper half distribution.



(d) Lower half/Upper half distribution on 64 cores.

The y-axis (vertical) represents the number of logical threads per core. The x-axis represents the code index in the cluster. The z-axis (depth) represents unfolding steps, i.e. time during execution.

Figure 13.9: Different logical thread distributions for QuickSort.

Figures reproduced with permission from [SHJ11].

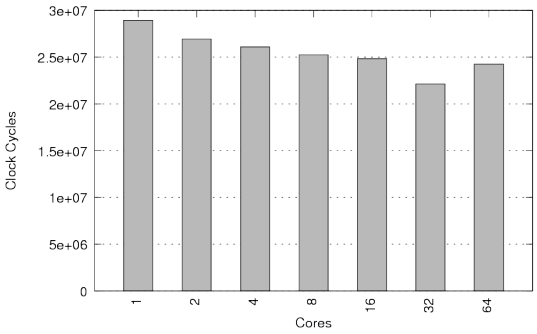


Figure 13.10: QuickSort performance using automatic load balancing.

Figure reproduced with permission from [SHJ11].

We considered that the higher latency of a two-pass transaction would still be relatively small compared to a software-based load balancing scheme, and would be compensated by a lower load imbalance. To enable the use of this feature in software we introduced the optional keyword "`sl__strategy(balanced)`" as a specifier for the constructs `sl_create` (Appendix I.5.8.1) and `sl_spawn` (section 6.3.5).

Analytically, this *automatic load balancing* feature is only beneficial to performance when the rate of new delegations over the entire local cluster is lower than the bandwidth of the delegation network for the cluster. Otherwise, contention occurs on the delegation network: the latency of each request increases and may not be compensated any more. The applicability of this feature can thus only be increased either by using a coarser concurrency granularity, or increasing the delegation bandwidth.

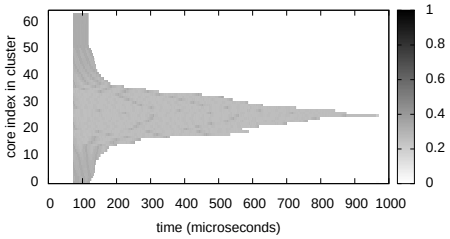
This feature was thus combined with concurrency throttling in the QuickSort example: the code was modified to use load balancing, and to perform sorting sequentially for sub-lists of less than 10 elements. As fig. 13.10 shows, this combination indeed enables improved performance up to 8 cores (cf. fig. 13.8 for comparison). In this benchmark, beyond 8 cores the increased latency is still not properly compensated by the workload on each core. It is possible to increase this threshold at the cost of more load imbalance.

13.5 Optimization of performance per watt

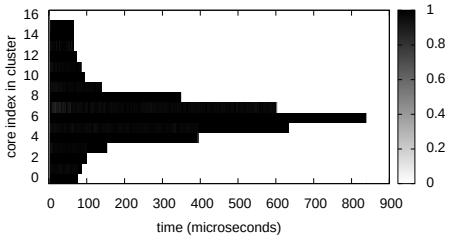
Another aspect illustrated by fig. 13.6 is the spectrum of possible parameter choices when selecting an implementation to meet a performance constraint: one can either tweak the number of thread contexts per core, the number of cores actually used or the load distribution of logical threads across cores.

When performance constraints are expressed as *real-time deadlines*, e.g. "this computation must complete in less than 1ms," there may still exist multiple parameters that satisfy the constraint. With the example from fig. 13.6, this specific deadline can be met using e.g. 4 cores with round-robin distribution and 16 threads contexts used per core, or 64 cores with even distribution and 1 thread context per core. We illustrate this diversity in fig. 13.11.

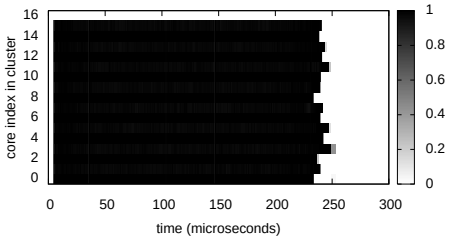
Although intuitively, a selection should favor a smaller number of cores to minimize energy usage, i.e. the parameters of figs. 13.11b to 13.11d over those of fig. 13.11a, the optimal decision strategy may not always be to choose the highest performance per unit of area. For instance, the best choice may depend on the availability of frequency scaling.



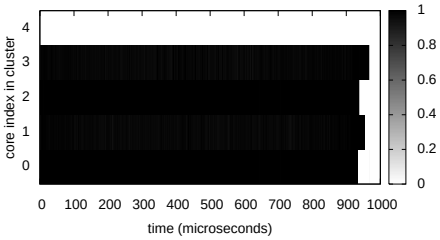
(a) Even distribution, 1 thread/core, 64 cores.



(b) Even distribution, 16 threads/core, 16 cores.



(c) Round-robin distribution, 16 thread/core, 16 cores.



(d) Round-robin distribution, 16 threads/core, 4 cores.

Figure 13.11: Per-core activity for the example heterogeneous workload with a 1ms deadline.

Without frequency scaling, parameters that perform with an overall load imbalance may be beneficial as they would allow the system to gate the clock of and power off cores that become unused over time. This would favor choosing e.g. the parameters of fig. 13.11b over those of fig. 13.11c. In contrast, if frequency scaling is available, the configuration of fig. 13.11c can be run at a third of the clock frequency and both configurations of figs. 13.11b to 13.11d may have a similar energy cost. The selection can then be guided by other considerations such as heat dissipation, which would then favor the configuration of fig. 13.11c which better spreads the load than those of figs. 13.11b and 13.11d.

▷ These considerations support our earlier remarks from section 11.3.2: we still lack a performance model which accounts for both configurable core cluster sizes and energy usage by computations. For this reason, a simple on-line resource manager that dynamically places computations based on resource availability and application demands still eludes us—despite, and perhaps regardless of, the availability of hardware primitives for concurrency management.

To summarize, the evaluation confirms that hardware microthreading as an architecture design direction can increase computing density (instructions executed per unit of time and unit of area), including for heterogeneous concurrency, but it does not fundamentally change the problem of making high-level scheduling decisions. If anything, it makes it computationally harder due to the larger amount of concurrency being managed.

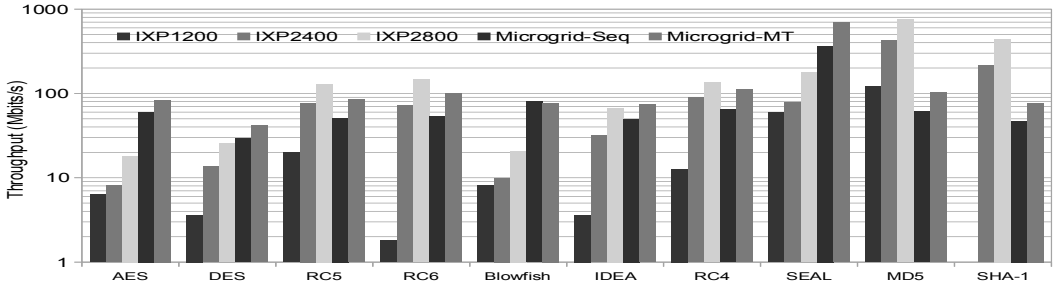


Figure 13.12: Throughput for one stream on one core.

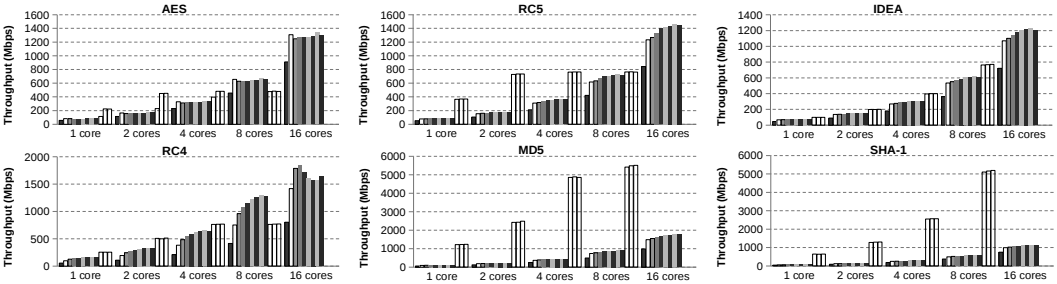


Figure 13.13: Combined throughput for 1-8,16 streams per core on 1-16 cores.

The figure shows 1-256 streams. The IXP2800 performance is shown in the leftmost 3 bars at each core group.

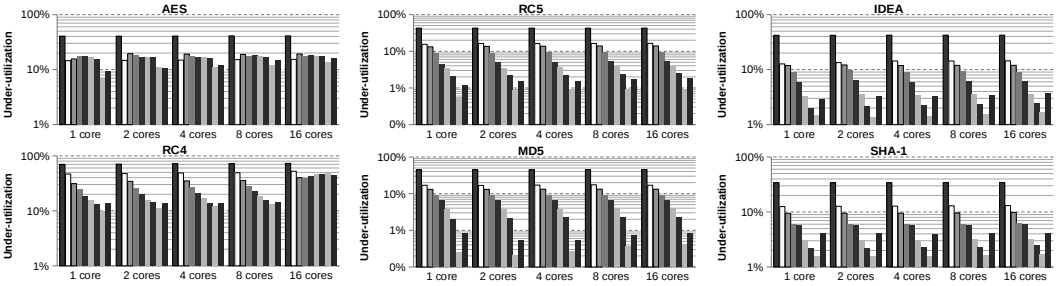


Figure 13.14: Pipeline under-utilization for fig. 13.13.

13.6 Applicability to throughput-oriented applications

The authors of [TLYL04, YLT05] have introduced NPCryptBench, a benchmark suite to evaluate network processors. We have run unoptimized code for these ciphers and hash algorithms on our reference platform. First the throughput of the unoptimized code for one flow on one microthreaded core is compared against the unoptimized throughput for one flow on one core of the Intel IXP chips ([TLYL04, fig. 4], [YLT05, fig. 3]). Two codes are used on our platform, one purely sequential and one where the inner loop is parallelised. Both are implemented using only our proposed interfaces from chapters 6 and 11. As the

results in fig. 13.12 show, the microthreaded hardware provides a throughput advantage for the more complex AES, SEAL and Blowfish ciphers, whereas the dedicated hardware hash units of the IXP accelerate MD5 and SHA-1. For the other kernels, the microthreaded hardware is slower: with RC5, RC6 and IDEA, a carried dependency serializes execution and minimizes latency tolerance. With RC4, the modified state at each cipher block must be made consistent in memory before the next thread can proceed, which also partly sequentializes execution. Further throughput deviation from the IXP should be considered in light of the frequency difference (1.4GHz for the IXP vs. 1GHz for our platform) and the fact the microthreaded hardware was not designed specifically towards cryptography.

Figure 13.13 shows the scalability of the most popular cryptographic kernels, using the purely sequential, unoptimized code for each stream on our platform and the Level-2 optimized code for the IXP2800 ([TLYL04, fig. 6], [YLT05, fig. 8]). For each sub-cluster size, increasing the number of flows per core increases utilization (fig. 13.14) and thus overall throughput. Throughput is furthermore reliably scalable up to 16 cores. With RC4 and 64 flows on 16 cores the workload reaches the memory bandwidth of the chip; with additional flows, contention on the internal memory network appears, and the utilization is reduced slightly as well as the throughput. The throughput then stabilizes at 96 flows around 1.6Gbps.

13.7 Issues of system bandwidths, design trade-offs

13.7.1 External bandwidth to memory

According to the Amdahl/Case rule of thumb on balanced designs, cited and updated in [GS00], a platform should provision 1 bit per second of external bandwidth for each potential instruction per second. With cores clocked at 1GHz this would imply 1Gbit/s per core. At first sight, this seems matched in the proposed configuration: the internal cache network supports 64-byte transfers at 1GHz, totalling 512Gbits/s internal bandwidth, and uses 4 DDR3-1600 external channels totalling 409Gbits/s external bandwidth, well above the 128Gbits/s implied by the rule of thumb.

Yet while carrying out evaluation activities with our community we did observe experimentally that computation kernels running on 1 to 64 cores, i.e. before fully utilizing the 128 cores available, can saturate the memory bandwidth. We reported on this in [BGH+11]. Further analysis with our peers in charge of the hardware design has revealed the following limiting factors:

- a *programming error* in the simulation infrastructure: although the DDR interface specification supports pipelining, this feature was not used in the cycle-accurate platform simulation used for evaluation. Instead, this implementation supported only one outstanding request to external memory. The maximum bandwidth for consecutive reads was thus determined by DDR's read latency (tCL), and not the issue delay (tCCD), yielding a maximum of 34Gbit/s per DDR channel in the reference configuration instead of the expected 102.25Gbit/s;
- even if full pipelining was available, *multi-thread interference* in Dynamic RAM (DRAM) accesses would still limit bandwidth. This is inherent to the low-level access protocol to DRAM banks: the time to load a row of cells into the *row buffer* is higher than the time to access different cells within the row buffer. While successive accesses by a single thread can be expected to target addresses within the same row, accesses by

multiple threads are interleaved on the channel and may require in the worst case to *switch DRAM rows* at each memory operation. This would incur DDR's row precharge (tRP), activate (tRCD) and minimum open-close latencies (tRAS) at every access, and thus limit the bandwidth for DDR3-1600 channels to 10.5GBit/s per channel.

This last issue is the most severe and has affected all the memory-bound results reported in this chapter and previous academic publications up to 2012. Under high load by heterogeneous multithreaded workloads, the *visible bandwidth* of 4 DDR3-1600 channels may degrade to 42Gbit/s, well below the 128Gbit/s recommended for a balanced design. To overcome this issue, the system designer may consider that the DRAM access delays only constrain accesses to a single bank. Additional bandwidth can thus be obtained by increasing memory-level parallelism, i.e. the number of visible banks, and *expose the address-to-bank mapping* to operating software so that it can map different activities to different banks.

This solution was identified in [JYS+12]. However, the authors of this paper explain that increasing memory-level parallelism has a non-trivial cost. With the advent of narrow point-to-point memory interconnects such as FB-DIMM [HV05, GJWJ07] and Intel's QuickPath Interconnect [Cor09, ZBMS10], it becomes possible to overcome the traditional package pin count limitation and *increase the number of separate memory channels*; however this comes at the cost of extra latencies and power consumption. Alternatively, one may want to *increase the number of independent banks per DRAM module*, however the authors estimate that market effects will prevent this opportunity and mandate increasing core count to bank count ratios in future systems. Instead, the authors suggest to use *DRAM sub-ranking*, which allows the memory controller to load data from different ranks into the same row buffer; however this comes at the cost of extra logic and latency per memory channel.

▷ Future work must thus determine the technology sweet spots that maximize external memory parallelism, and thus the visible bandwidth, at a given logic and energy cost. The *memory topology must be exposed* at the machine interface and the operating software must use this information to map different software activities to different DRAM banks.

13.7.2 Internal bandwidth

The overall *internal bandwidth of the on-chip networks* constrains the communication patterns of distributed algorithms and the minimum latencies of SIMD/SPMD operations. As such it is a factor in the maximum performance that can be reached for a given workload.

Any given choice of platform parameters will yield a specific set of network properties. For example, the memory network in the reference configuration has a theoretical point-to-point maximum bandwidth of 512Gbit/s, and the delegation/distribution NoC has a maximum point-to-point bandwidth of 8Gbit/s. Furthermore, any choice of protocol will impact the *visible internal bandwidth* of the chip. For example, the proposed distributed cache network uses update and eviction messages to propagate stores across caches. These messages reduce the bandwidth available to other messages like loads. In [BGH+11] we discovered that inter-cache management messages can cause a reduction of up to 40% of the visible on-chip memory bandwidth.

▷ We then considered whether this result is a suggestion that the chip designer should provide wider network lanes and higher connectivity, and/or whether new protocols should be designed with more control to software to avoid unnecessary traffic. However, any investment of logic into the network would reduce the maximum number of cores, and thus possibly become detrimental to the performance of compute-bound workloads. This should remind

us of the discussion in [EBSA+11], where the authors argue that for any given many-core chip configuration a significant area of logic will be under-utilized, invisible to most workloads except the few that require it; this is subsequently called *dark silicon*. Addressing this issue may involve the integration of re-configurable logic on chip, where a resource manager on the chip would configure gates towards either extra network lanes (higher communication bandwidth) or cores (higher computation throughput) depending on the workload. We are not aware of any existing research in this direction at the time of this writing, and this issue may constitute a basis for future research.

□ In the mean time, we should highlight here that any *quantitative analysis* of the high-level issues about the internal parameters of a chip cannot occur before specific chip parameters are selected. We can thus argue that the *crystallization* of a platform, such as performed in this book, constitutes a necessary first step towards the development of actual CMPs around hardware microthreading.

13.8 Relevance of thread-to-thread synchronizer sharings

We have explained in section 4.3.3.3 that the ability to share physical synchronizers between multiple threads opens the opportunity to daisy-chain logical threads by overlapping the visible window of adjacent thread contexts and spreading the logical indexes in a round-robin fashion.

This feature was originally proposed to explore whether bulk created microthreads are a suitable alternative to dependent sequential loops. The general idea is that expressing a sequence as a network of dependent threads allows dependent threads from a loop to execute in parallel and reduces the need for hardware logic that "discovers" concurrency from the instruction stream. It was implemented in our platform, and we exposed language constructs to use it in chapter 6. An example benefit of this feature can be seen above in fig. 13.12. In these results, a *single stream* of data flows through a cryptography kernel implemented both sequentially and using dependent microthreads for the inner loop. As illustrated in the figure, using dependent microthreads increases the performance of most kernels by 40%-90%.

Yet further analysis by [SEM09] suggest that the feature has only marginal benefits. It only enables performance gains when both the following conditions are met:

- the loop has a constant stride;
- all carried dependencies fit in synchronizers (note that their number is limited due to the substrate's ISA register naming in the instruction format—our parameters from chapter 8 further limit their number to 6).

Furthermore, the authors of [SM11] propose to rewrite automatically data-parallel workloads that would require a dependent loop in sequential code using a regular dataflow graph that would be efficiently executed using a specialized software scheduler running on multiple microthreaded cores. This transformation has been prototyped successfully in a compiler. As illustrated by fig. 13.2 ("Parallelised C"), this scheme obtains performance figures close to the hand-optimized code.

□ Beyond the software scheduler of [SM11], we have identified other general ways to organize partially sequential computations between microthreads using only *existing features of the design*:

- plain *sequence* between logical threads on each core can be requested by setting the "block size" parameter from section 4.3.2.2 to 1;
- any reduction using commutative and associative operations *on one core* can be implemented using only the "global" synchronizers shared by all thread contexts participating in a bulk creation:
  - simple operations that require only one machine instruction can be used with a global synchronizer as both input and output operand. The dataflow scheduler will automatically serialize all such instructions in the pipeline, possibly out of logical index order;
  - complex operations that either require multiple machine instructions, or involve updates to memory which must appear atomic, can be enclosed by a transaction started with a "clear" operation to one of the global synchronizers, and ended with a "set" operation to the same synchronizer. The clear operation both reads from its operand and sets its dataflow state to "empty." Once the first clear executes, all further clears from other threads will suspend until the synchronizer is set again. The set is then performed by the thread that has executed the first clear at the end of its transaction. This wakes up another thread, which is given a chance to run "clear" and open a new transaction, and so on.
- any reduction using associative operations *on multiple cores* can be implemented using successive bulk creations of thread functions performing a *parallel prefix scan* [LF80, SHZO07] of the data to reduce, or distribute local reductions as described in section 11.2.4;
- efficient multi-core reductions using multithreading on each core for latency tolerance can be achieved by combining the techniques above.

We have tested these opportunities and confirm anecdotically that they yield high core utilization. Moreover, using global synchronizers for reductions further reduces register file utilization, since the global synchronizers are only allocated once. To simplify the use of global synchronizers for reductions in programs, we further suggest to introduce Cilk's *hyperobjects* [Lei09] as a low-level language feature in our interface language, upon which further abstractions can be constructed; we are planning this implementation as future work.

▶ To summarize, we have not yet found a decisive benefit of the "shared synchronizers" pattern; instead we observed that its intended use cases can be addressed using other, simpler features. Also, as illustrated in figs. 13.13 and 13.14, interleaving different purely sequential activities on the same cores provides another means to maximize utilization of each core. These observations may indicate that the "shared synchronizers" feature was an instance of premature optimization. We could not conclusively support the need for this feature; future work may even suggest to remove it from implementations to further simplify the microthreaded cores and their conceptual model. Our careful separation of its description in chapter 4, which isolates this feature from the more interesting aspects of the architecture, was an intentional step in this direction.

Summary

In this chapter we have shown that the provided platform support was sufficient to carry out evaluation via reduced, yet fully functional benchmarks. Regardless of the specific performance figures, the fact that evaluation was possible using traditional engineering workflows is a sign that the platform "looks and feels" like traditional platforms sufficiently for integration in existing evaluation methodologies.

Furthermore, by looking at the evaluation results we show that the proposed tool chain was appropriate to utilize efficiently the processing abilities of the new architecture for most data-parallel workloads, and obtain higher performance densities than contemporary chip architectures. As predicted in section 5.3, our technology empowered both our community and ourselves to step out of the engineering effort and gain high-level insight over issues of concurrency management and chip design, briefly exposed in this chapter.

Chapter 14

System-level issues
—Lessons learned on and about the way to integration

Abstract

During interactions with an audience, the platform provider may gain additional insight about the innovation from the interaction process itself, independently from the audience. In this chapter, we provide an example of this. During our implementation work for the platform in part II, we discovered additional *system-level* issues not directly visible to our audience when carrying out the evaluation work reported on in chapter 13. Again, as predicted in section 5.3, these issues translate directly to feedback about the architecture design. We document them here.

Contents

14.1 Mutual exclusion

To manage shared resources, the platform must provide a means for the operating software to negotiate *atomic transactions*, or mutual exclusion, over these resources from the perspective of concurrent activities.

In the case of physical I/O devices that have specific locations on the chip, mutual exclusion can be achieved by forcing the serialization of work at the locus where the physical interface exists. For example, in our work, the processors physically close to the I/O devices feature a unique thread context whereby remote requests to create work at that context would not be accepted until the previous work has terminated. This is equivalent to the "secretary" concept from [Dij71, p. 135], and transparently reuses the existing mechanisms from chapters 3 and 4.

To control access to *logical* shared resources like heap allocators or logical queues, existing software requires the availability of *logical synchronization devices* like monitors, locks or semaphores. How these are *implemented*, whether via memory transactions over a cache coherency protocol, or dedicated synchronization messages on a NoC, or a combination of both, seems to us orthogonal to the architecture design principles from chapters 3 and 4. Nevertheless, we were asked to make an educated commentary about the opportunity to implement *all* logical synchronization devices using only the hardware-based serialization mechanism described above. The motivation for this is to alleviate the need for global coherency on the memory network (cf. section 3.4.1).

We can comment on this opportunity with two observations.

The first observation is that an efficient implementation of a back-propagation mechanism of a contention situation to the locus of issue requires attention throughout the chip. To understand why, consider first that proper *suspension* of a thread, in a manner that lets other threads progress, requires to store the identity of the thread *somewhere*, i.e. in a dedicated memory structure with a physical location. This state must then wait *passively* for a wake-up event. This implies that a thread can only be woken up upon reception of an explicit message by the locus where its continuation is stored. In the mutual exclusion scenario, if a thread attempts to access a shared resource already in use, it must suspend; conversely, when the shared resource is *released*, the resource must notify all physical locations that contain suspended threads waiting on the shared resource. For example, with the physical serialization outlined above, threads must suspend upon upon writing a creation request over virtual channels to a contended shared resource; when the shared resource becomes available, it must wake up, via flow control over all virtual channels to the resource, the remote threads that were previously blocked.

A potential naive implementation of this necessary notification is based on the traditional concept of "waiting thread queue," where each shared resource would store locally a dynamically evolving set of client loci containing thread continutations to be resumed upon release. This would in turn require a local memory whose size is linear with the total number of loci in the system which may participate in the mutual exclusion. If any core must be able to host a shared resource, this implementation would require a space requirement quadratic in the size of the system. Instead, to minimize the overhead, a tree-like notification network of virtual channels can be built from the shared resource to all candidate clients, using flow control to control access to the shared resource. This in turn requires implementation of flow-control fan-in at all branching points on the NoC with a prioritization scheme to avoid starvation, multiplied as necessary if any core must be able to host a shared resource.

The trade-off is therefore a choice between increased storage costs at each locus, or a coordinated protocol between all loci which must in turn be made resistant to network contention and faults, even when these originate from loci not participating in the mutual exclusion.

The other observation is that it forces *every* access that *queries* the state of the shared resource, even when it is not contended, to place a roundtrip across the NoC on the critical path through execution. The latency of this roundtrip *may become arbitrarily large depending on other activities on the NoC*, even if the target shared resource is not contended. This violates a common expectation about logical synchronization devices, that they should have a *local* latency when they are not contended[1] [MCS91]. To avoid this situation, dedicated Quality of Service (QoS) protocols must be used on the NoC, in turn increasing implementation costs.

▶ To summarize, support for logical shared resources requires support from the architecture to negotiate mutual exclusion. Whether new dedicated hardware solutions are designed, or existing synchronization mechanisms over cache coherency protocols are reused, any solution must be *scalable*: not incur interference between unrelated activities and avoid mandatory overheads for non-contended resources. Beyond our suggestions from section 13.8, the corresponding mechanisms are not yet clearly defined in the architecture design from chapters 3 and 4 and must thus be addressed in future work.

14.2 Event and fault management

As soon as work starts to define a full system, the need arises throughout to handle *unexpected conditions*. Here we exclude software exceptions that can be implemented cooperatively using software techniques only, and those hardware faults that can be hidden from software entirely (via redundancy, hardware error correction, etc.). We further consider separately *traps*, which are *caused* by programs as an effect of performing particular actions (e.g. divide by zero, address translation fault) and *asynchronous events* which are not expected to occur at specific points during execution, but are still expected to occur *sometimes* (e.g. availability of out-of-band input on a communication channel, termination of a child process). The latter must be supported in particular before a system can be advertised as an *interactive* general-purpose platform (cf. section 1.2.1).

Traditionally, the common mechanism used to report both traps and asynchronous events is to *interrupt* an instruction stream, and transfer control of its execution preemptively to a configurable *event handler* which may then decide to either address the exception and transfer control back to the program, or trigger an alternate behavior, including possibly program termination. In contrast, in a parallel setting with potentially many threads of execution, a diversity of solutions may be considered.

One alternative, used in the POSIX threading API, is to report traps via interrupts to the specific thread that causes them, and report asynchronous events via interrupts to one, non-deterministically chosen thread among those currently executing threads that have declared their readiness to be preempted via `sigaction` and `sigsetmask`. There are two known shortcomings to this approach that limit its scalability to larger parallel systems. One is that every event in the system, including non-local communication events, must encode the identity of its causal thread somehow so that an error can be reported accurately.

[1] e.g. the time to acquire a lock that is not previously locked, or to decrease a semaphore with a counter still greater than 1, should stay minimal and independent of the overall system activity.

While this is cheap to achieve for FPU exceptions, it may cause a burden on the hardware with e.g. accesses to virtual memory when address translation is resolved by a common MMU shared among many cores, or accesses to I/O channels when events may be detected topologically far from the locus of thread execution. The other shortcoming is that this form of reporting requires every thread to have access to an area of storage where its current state can be saved at the point the signal handler is invoked. As we have explained in chapter 9 this is a non-trivial requirement as the number of threads grows.

There several other alternatives possible.

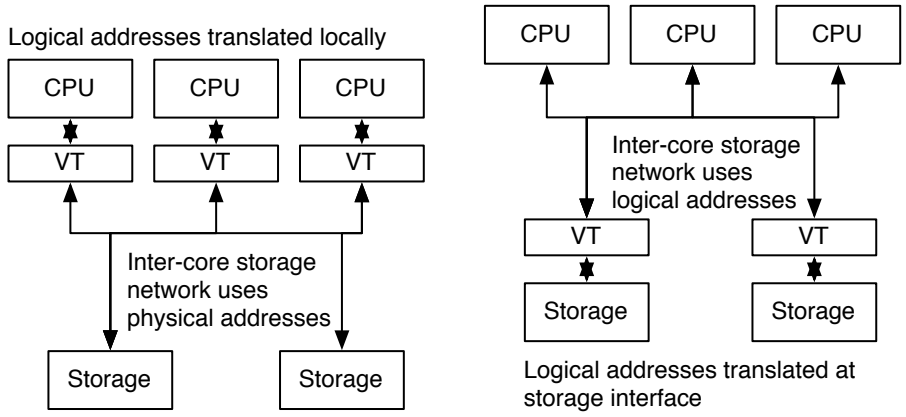
The designers of UTLEON3 propose in [DKK+12] to stop the execution of all currently running threads, and trigger the execution of a high-priority "handler thread" that would run to completion before execution of the other threads resumes. This mechanism was also suggested earlier by the author of [van06]. This mechanism is intuitively appropriate to handle asynchronous events, or to cause termination upon traps, but it does not let software react to salvageable traps such as floating-point exceptions. In [van06], we can find an extension of this solution which proposes to:

- report traps to their causal threads by suspending the causal thread, triggering execution of a separate, *fault handling thread* on the same core, and have the fault-handling thread automatically receive from the hardware, as input parameter, the identity of the causal thread. Upon resolving the behavior, the fault-handling thread could then choose between resuming execution of the causal thread or trigger some alternate behavior like program termination.
- for asynchronous events, simply create new, regular threads for each event received.

Both with this solution and the UTLEON3 solution above, the reason why interruption of the control flow *cannot be used* is that the visible synchronizer window layout (cf. sections 3.3.3 and 4.2 and chapter 8) expected by the fault handler may not be compatible with the window layout of any of the currently executing threads. Note however that this physical separation between causal threads and fault handling threads does not imply that the fault handler cannot impact the control flow of the causal thread: the hardware interface could provide a primitive to the fault handler to override the PC of the causal thread and write its registers remotely. This would in turn be sufficient to allow individual threads to register different exception behaviors in software, and implement e.g. C's `longjmp`, which is the traditional mechanism to override control flow from asynchronous handlers.

A more interesting issue related to the creation of new threads is that existing software frameworks often assume a consistent view of all the program's variables when an event handler is invoked. The corresponding issues from chapter 7 notwithstanding, with a large many-core system this expectation would require global synchronization of the memory state at every event delivered, in turn significantly impacting performance. While global synchronization could be envisioned for rarely expected faults, it seems too expensive to require for general asynchronous events such as notifications for external input.

Researchers in operating system and language design may see here an opportunity to redefine the software abstractions for handling unexpected events, including abandoning the vocabulary related to "interrupts" and "control flow preemption." A future-oriented solution may include abandoning precise exceptions and global view on the program state altogether, and acknowledging that the physical location of the execution of an event handler on chip will impact how much of the program's state it can observe. However, embracing this conceptual revolution so early would be ill-advised, since, as we explained in chapter 5,



(a) Address translation occurs at the point of issue.

(b) Address translation occurs close to the storage.

Figure 14.1: Possible locations for the address translation logic on chip.

integration in existing ecosystems requires solutions that preserve current abstractions, at least in an initial phase.

□ Instead, we propose as an alternative approach to exploit the companion processors that we introduced in section 5.5.1. For this, we first differentiate between programs that explicitly manage exceptions, recognized by their explicit use of the `signal` and `sigaction` APIs, from those that don't. The programs that use the API are then constrained to always execute on the companion processors, where their threads can be interrupted in the traditional way. The programs that don't use the API are not constrained and can be mapped anywhere on the chip; any system exceptions they receive are then handled in the way compatible with the default POSIX behavior, i.e. either by ignoring them (e.g. `SIGIO`, `SIGURG`) or by causing termination of the entire program (e.g. `SIGINT`, `SIGSEGV`). This solution preserves past assumptions for programs that exploit them, while letting assumption-free programs take advantage of all the architecture's features. However, we did not evaluate this solution experimentally and cannot report on its implementation costs.

▶ Regardless of which solution is eventually adopted, we highlight that the mechanisms for handling unexpected events must be duly documented and illustrated before the proposed architecture can be used as a general-purpose substitute by its envisioned audience.

14.3 Virtual addressing trade-offs

In the proposed target ecosystem from section 5.4.1 and others, support for *virtual address translation* is expected from the hardware to implement storage virtualization and process isolation.

We were asked to make an educated commentary about the spectrum of choices about *where on chip to perform translation*, which we illustrate in fig. 14.1: either translation can occur at the locus where memory requests are issued, i.e. close to the cores, or it can be centralized and shared between multiple cores. Centralization is possible because, with the generalization of 64-bit wide addresses, the address space may be sufficiently large to host

the data manipulated by all concurrent activities in the system. Meanwhile, isolation can be provided using capabilities [JLI98] over regions of the shared address space. This is the view taken by Opal [CBHLL92], Mungi [HEV+98], Mondrian [WCA02] and other Single Address Space Operating Systems (SASOSs). For example, in our work, the implementations for the design from chapters 3 and 4 perform address translation at the chip boundary with a distributed MMU, one per DDR channel. We comment with two observations.

The first is that most target ecosystems, in particular those identified in section 5.4.1, have historically established a strong conceptual binding between process boundaries and virtual address spaces, for example the expectation that each process has its own "private" address space to storage. To preserve this mindset, a system implementation that shares translation management between multiple cores, such as the platform considered, must either:

- ensure that processes with distinct address spaces are segregated to separate locations on the chip, so that the physical source of a memory request is sufficient to distinguish its originating process, or
- explicitly communicate with every memory request the identity of the issuing process.

The former approach would be detrimental to utilization and performance when there are more processes active than translation domains, even if the total number of processes is smaller than the number of cores. The latter approach is costly in either hardware logic or performance as larger mapping caches (e.g. TLBs) will be needed to support larger numbers of processes. These trade-offs are analogous to the opposition between virtual vs. physical *cache* indexing discussed in [CD97a, CD97b].

The other observation is that any choice as to where address translation occurs must be combined with a choice as to where process boundaries for isolation are checked. Process boundaries are typically checked by system services, such as resource managers, to ensure that separate processes obtain separate views on the service. In presence of virtual address translation, each entity that checks process boundaries must be able to look up data in memory *on behalf* of a requesting process, i.e. relative to the private address space of the requesting process. This in turn implies that system services are also "clients" to translation and participate in the design trade-offs identified above.

▷ To summarize, we cannot determine whether centralization on the chip of address translation and isolation checks can yield significant benefits for the target ecosystems without looking quantitatively at the number of separate processes that would be defined. Nevertheless, we are able to argue that address translation and the definition of process boundaries must be handled in the same design discussion, because the system services that check process boundaries are also clients to address translation.

14.4 Implicitly carried state

The standard C library, like most traditionally sequential APIs offered in existing general-purpose systems, depends on *implicitly carried state* invisible from programs. We detail this observation below in sections 14.4.1 and 14.4.2, as well as its likely impact on system design in section 14.4.3.

State	Carried by	Reset by	Averted by
Set of opened file descriptors	`open`, `close`, `exit` (system)	N/A	N/A
File positions	`read`/`write`	`seek`, `lseek`	`pread`, `pwrite`, `aio_*`
Set of opened streams	`fopen`, `close`, `exit` (C library)	N/A	N/A
Status of standard streams (`stdin`, `stdout`, `stderr`)	`puts`, `printf`, `getchar`, `scanf`, etc.	N/A	Explicit stream operands (`fputs`, `fprintf`, etc)
Error status (`errno`)	All APIs related to system interfaces	(program)	N/A
`dtoa` allocator's internal state	`dtoa`, `(f)printf`	N/A	N/A
`malloc`'s internal state	`malloc`, `free`, `realloc`, `calloc`	N/A	Anonymous `mmap`
Set of memory regions visible by program	`mmap`, `sbrk`, `shmat`	N/A	N/A
Environment variables (`environ`)	`getenv`, `setenv`	N/A	N/A
Registered destructors	`atexit`, `exit` (C library)	N/A	`_Exit`
Floating-point environment	Math functions	`fesetenv`	`#pragma STDC FENV_ACCESS OFF`
`strtok` buffer	`strtok`	`strtok`	`strtok_r`, `strsep`

Table 14.1: Implicitly carried dependencies in C/POSIX APIs.

14.4.1 Problem statement

The assumption of sequential program execution has traditionally enabled API designs with implicitly carried state from one API call to another. For example, POSIX's `read` and `write` calls carry the current I/O position in the file implicitly from one call to the next. Due to their history as an environment for mostly sequential programs, both the C language, the POSIX interfaces and other API framework for general-purpose programming languages are rife with sequential APIs; we list the most common from Unix in table 14.1.

There is no theoretical difficulty here: a given sequential programming interface is either known to be appropriate for concurrent calls (this is documented as "reentrant" or "thread-safe"), or it is not. Practically, this difference determines who is *responsible* for ensuring proper synchronization: in the former case, the service provider must provide synchronization, and in the latter case the application programmer must ensure that calls are synchronized. *Race conditions* occur when neither take this responsibility, and *over-synchronization* occurs when synchronization is negotiated on both sides of the service interface.

□ What we found interesting however, is the impact of this *semantic mandatory sequentialization* on multithreaded performance: regardless of the overall structure of the program, how it organizes concurrency between components, and how API services are implemented,

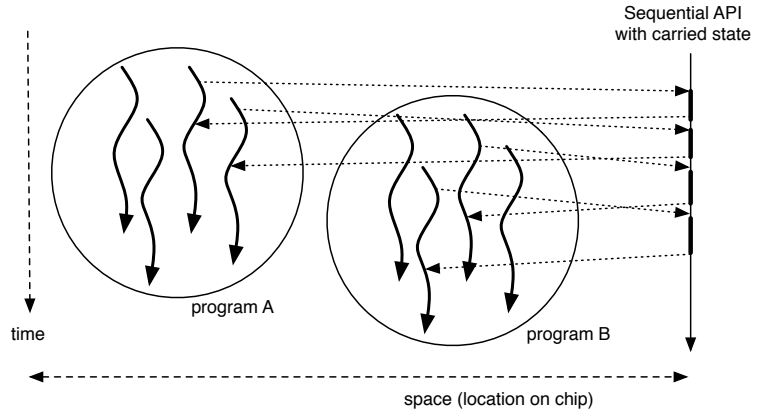


Figure 14.2: Forced synchronization through sequential APIs.

the set of all effective API calls to sequential APIs during execution constitutes an *implicit sequential path*, invisible from the explicit program description.

We illustrate this in fig. 14.2: even when threads of distinct application components are independent from each other, an implicit synchronization occurs every time they issue a call to a common API. The API service, as a participating entity during the execution, is in fact a *shared resource* which can cause contention.

In general terms, the implicit sequential path of a sequential API becomes part of the *critical path* of execution, and thus a constraint on performance, if the overall *issue rate* of API calls across all threads to the API provider becomes equal to the *maximum bandwidth* of the API provider.

14.4.2 Contended resources

In our setting, the large amount of concurrency available in hardware allows us to increase the issue rate to a sequential API far beyond the bandwidth of single cores where the API runs. We observed concrete occurrences of contention caused by sequential APIs in the following circumstances:

- dynamic memory allocation: calls to the `malloc` and `free` APIs are frequent, for example in code from higher-level programming language like SAC [GS06]. They cause contention if served by a single heap allocator. We were able to reduce this effect partially by using a concurrent heap allocator, however larger numbers of concurrent heaps in turn cause contention on system-level dynamic allocation of physical RAM pages to back the virtual address space;
- memory reclamation: to enable garbage collection of unreferenced arrays in SAC programs, we developed jointly with our project partners a deallocation strategy based on reference counting. This strategy uses a system service for mutual exclusion to protect concurrent accesses to the reference counters. In this work, we found that the set of updates issued by independent threads to single reference counters cause contention on highly parallel workloads. We document this in [HJS+11];
- dynamic processor allocation: to enable dynamic placement of application components to processors, we implemented an allocation API for processors that programs can use

to reserve and release entire clusters of processors at once, using the abstractions introduced in [JPvT08] and chapter 11. Since this allocator must keep a map of which cores are allocated as implicit state, we also found that it becomes a source of contention on dynamically heterogeneous workloads.

In contemporary applications running on other platforms, this source of contention is usually averted in either of three ways:

- either the issue rate (average number of calls per unit of time of all threads) is kept low, by either having few threads running overall or concentrating most API calls in a "control thread"; or
- *process boundaries* are introduced between application components, so that the sequential APIs in each process can be served by separate service providers in operating systems (for example, operating system kernel components running on separate processors); or
- programs avoid using sequential APIs altogether, substituting uses of distributed services instead.

▷ We highlight that explicit control of API issue rates is impractical in most programming languages and that process boundaries increase the communication latency between application components. Therefore, any new system implementation which aims at providing a general-purpose platform must provide contention-less equivalents for *at least* these services listed in table 14.1, for these are the services in pervasive use across most current applications and language run-time systems. Here we would like to suggest further that *major abstractions in existing operating systems will be altered during this process.*

To support this statement, we take the example of file I/O. The most intuitive carried state of `read`/`write` is the file cursor at which I/O operations are effected. It is possible to remove this carried state as in `pread` and `pwrite`, which take the file position as an explicit argument. Unfortunately, the *file descriptor* remains: every POSIX call which manipulates file descriptors must check *whether the descriptor is valid* and *which object it designates in the virtual file system*; this mapping of descriptors to objects is a carried state updated by the global sequence of calls to `open` and `close`. To remove this state, it is necessary to design the file access API to use *stateless file descriptors*, which do not need to be explicitly opened/closed. Although this feature is yet rarely used directly in applications, it is already available in popular distributed file systems (e.g. Sun's Network File System (NFS) [SGK+85], Hadoop [SKRC10]) and we predict its increased use in disfavor of Unix' traditional numeric file descriptors. This may have a dramatic impact on other APIs, such as `select`, or otherwise program algorithms which assume descriptors numbered contiguously from 0.

14.4.3 General problem and consequences on hardware design

While single-machine operating systems with a distributed internal structure are a mature and well-studied field of research [TVR85, TR86, SJ96, TS06], we found that the focus of most previous research on internal system data structures was on state isolation for security, both to increase accountability and fault tolerance. Comparatively, little research has been carried out to optimize performance via distribution within operating systems until recently [SPB+08, WA09].

Instead, we found previous work on performance scalability at the level of networked application services. An exhaustive and comparative review of application-level schemes are available in [Fie00], whose author revolutionized the exploitation of applications over the Internet by popularizing REST protocols: client-server, stateless, cacheable, and layered. Modern distributed file systems provide REST protocols; we suggest that future machine level services in operating systems should feature REST protocols as well to implement logical, software-only services.

The limit of the REST vision, however, is reached with *shared physical resources* such as address spaces in storage or external I/O ports, because the physical reality of their implicitly carried state cannot be moved around the system via logical messages.

▷ We can predict a *mandatory clustering of physical resources* as a consequence. As the amount of general-purpose concurrency in application increases, so does the concurrent use of system services; a given number of effectively parallel threads of execution will thus necessarily translate to a lower bound on the required bandwidth from system services providing access to shared physical resources. Since the energy and memory walls apply to system services as they do to applications, the only way forward to reduce contention is parallelism, i.e. multiply physical resources (memories, I/O connections, accelerators) and bound the visibility of each resource to processing elements that are topologically close. The resulting system structure is a *constellation* topology where each cluster has a limited amount of processing elements sharing common system services, with little to no visibility to the system services of other clusters. This vision has been reached independently, with the same justification, by the authors of [WA09].

14.5 Process state, termination and reclamation

While working on the implementation of system services as per chapter 5 and section 6.4.3, we struggled with *process termination*. The problem stems from the semantics of the `exit` system service and termination signals as triggered by POSIX's `kill`, `abort` or C's `raise`: when these events are triggered by a program, the run-time environment must terminate the program and *reclaim its resources*. Since these events can be triggered by one thread while other threads in the same process are active and running, a mechanism must exist to stop the execution of all threads preemptively and reclaim all memory and other system resources that were allocated so far.

Next to process termination, we also struggled to define *process data encapsulation*. Process-specific data in C comprises all mutable objects defined by programs and library code in the global scope, the shared heap(s) and thread stack(s). The most often used process-specific data from library code are the standard I/O streams (`stdin`, `stdout`, `stderr`), and the error status code variable (`errno`). While it is possible to encapsulate these variables behind accessors, as in the BSD code which substitutes all uses of the word "`errno`" by a call to a function "`__error()`," the implementation of these accessors itself must have access to process-specific variable instances. The issue we faced was how to *look up* which instance to use from a given thread, as the proposed machine interface does not provide the notion of process identity.

▶ To summarize, the need to support process data encapsulation as well as resource reclamation upon program termination places two requirements on partial hardware support for concurrency management, in particular the design from part I:

- mechanisms must be available to bind allocated resources, e.g. memory and I/O channels, to *process identities* which can be used to look up process-specific data and delimit

the extent of resource reclamation upon termination. Our proposal to introduce virtual resource identifiers in the TLS addressing scheme (cf. section 9.4.2.3) makes a first step in this direction; however, a fully general solution requires an interface to bind thread contexts to process boundaries and communicate automatically process identities to system services upon resource allocation events;

- once process identities are available, mechanisms must be available to both inquire from any thread the identity of the surrounding process, and to asynchronously terminate all concurrent activities belonging to a given process.

14.6 Feedback from education activities

Our framework was further used during various education projects at undergraduate levels. Using our proposed interface language and the separately provided full-system emulation, students have implemented various applications (Conway's Game of Life [Gar70], sorting algorithms, image processing filters, compression algorithms, signal processing, Scheme interpreter, and others) and demonstrated the effective use of single-core multithreading and multi-core execution to accelerate performance with relatively naive source code. The overall conclusions from this work regarding the evaluation of the architecture are similar to ▷ those outlined in chapter 13; however, we gained additional experience from working with students:

- comprehensive support for a large subset of the standard C library and POSIX interfaces is a strong prerequisite for students who have little previous exposure to operating system and language design. Indeed, most students seem to have troubles distinguishing whether program constructs are part of the language, the standard library, or the system interfaces. When the focus of a teaching activity is the introduction of new language features (e.g. concurrency constructs in our case), diluting the students' attention by forcing them to also deconstruct and re-learn C becomes detrimental to their success;
- similarly, a comprehensive troubleshooting infrastructure is essential to keep teaching activities focused, especially to avoid wasting student time with programming errors in the framework or conceptual issues in the new architecture design. Our choice to mandate serializability in the language semantics (cf. section 6.2.4) helped by allowing students to test and troubleshoot their programs independently from the new architecture. However, student progress was hampered by the lack of run-time program introspection tools to visualize trade-offs related to load balancing and scheduling. They also missed troubleshooting tools to help recognize false assumptions, in particular those about the memory consistency model.

To summarize, exposing a new platform to students during teaching activities was an effective way to exercise the tools and increase their accessibility. However, teaching activities confirm the usefulness of conceptual backward compatibility, such as introduced in chapter 5, since it is educationally difficult to introduce simultaneously new programming abstractions and a new vision on the system stack.

Summary

Although our work was taking the perspective of existing and legacy operating software, we were able to recognize some general issues relevant not only to the envisioned audience:

- mechanisms for mutual exclusion: this must be addressed by provisioning and advertising dedicated mechanisms in hardware;
- unexpected events and their reporting to programs: this must be addressed by introducing support for asynchronous notifications;
- the cost trade-offs related to the choice of a centralized MMU for address translation: these must be addressed by the system integrator in a dialogue, taking into consideration the environment where the system is to be used;
- performance bottlenecks due to stateful system services: this must be addressed by provisioning sufficient bandwidth to these services and evolve API designs to expose more concurrency;
- process-specific data and resource reclamation upon termination of programs: this must be addressed by introducing support for process identities;
- troubleshooting and inspection facilities must be provided to aid learning audiences in understanding the run-time behavior of their applications.

These issues must be further researched in future work before the design can be advertised as a general-purpose substitute to any of its candidate ecosystems.

Chapter 15

Conclusions and future work

—on the interplay between inner and outer questions.

> It is pitch black. You are likely to be eaten by a grue.
>
> Dave Lebling, *via Zork*

Contents

15.1 Design principles and opportunities

When the design principles behind hardware microthreading were first proposed, the envisioned features with the largest impact on performance were the exploitation of dataflow synchronization over instruction operands to implement dynamic scheduling, the acceleration of thread creation and synchronization in hardware via elementary machine instructions, and the unification of single-core and multi-core thread management under a common protocol also supported in hardware [BJM96, BHJ06a]. The more concrete CMP architecture described in part I constitutes a subsequent effort to consolidate and integrate these features so as to enable their empirical evaluation.

As we discovered, the principles of hardware microthreading impact the hardware-software interface beyond the extension of registers with dataflow state bits and hardware support for thread creation and synchronization. In particular, it the following extra features are visible from software:

- *configurable register windows*: the software can choose, upon the activation of a hardware thread, how many physical synchronizers (registers) are visible from the thread. This departs from previous ISAs where the number of registers is an architectural constant. The purpose of this change is to provide more efficient use of the register file by heterogeneous workloads.
- *active messages and asynchronous notifications via thread creation*: the reception of an external, asynchronous event by a processor core triggers the creation and activation of a new hardware thread, instead of the interruption of the control flow of an existing thread. This departs from previous ISAs where asynchronous events are delivered via interrupts. The purpose of this change is to simplify the individual core pipeline.
- *hardware synchronization services separate from main memory*: the extension of registers with dataflow state bits, together with the integration of a dedicated delegation/distribution NoC with the hardware scheduler on each core via the TMU, provides dedicated hardware support for I-variables, named critical sections, locking, barriers and semaphores. This departs from previous ISAs where synchronization semantics were provided as side-effects of ordering constraints on load and store operations to main memory. The purpose of this change is to provide lower synchronization overheads and a higher execution efficiency.

On their own, these extra features do not fundamentally interfere with existing ISA semantics. The system programmer can configure the register window of all threads to use a fixed number of registers, at the cost of a lower register file utilization, to obtain the fixed window semantics of previous ISAs. The designer of individual cores can integrate hardware microthreading in an existing RISC core without removing the interrupt delivery logic to preserve the preemption semantics of previous ISAs. The chip designer can integrate hardware microthreading in a CMP without removing support for consistency control in the caches and the memory network, to preserve the memory-based synchronization semantics found in existing CMPs. In theory, there is thus space in the design spectrum for a new CMP constructed from legacy RISC cores with hardware microthreading integrated as an extension. We depict this opportunity in fig. 15.1.

However, the researchers at the University of Amsterdam have made one further design step before investing in an implementation. They observed that asynchronous event delivery via thread creation is redundant with support for interrupts, and they observed that a dedicated synchronization protocol in hardware is redundant with memory support for

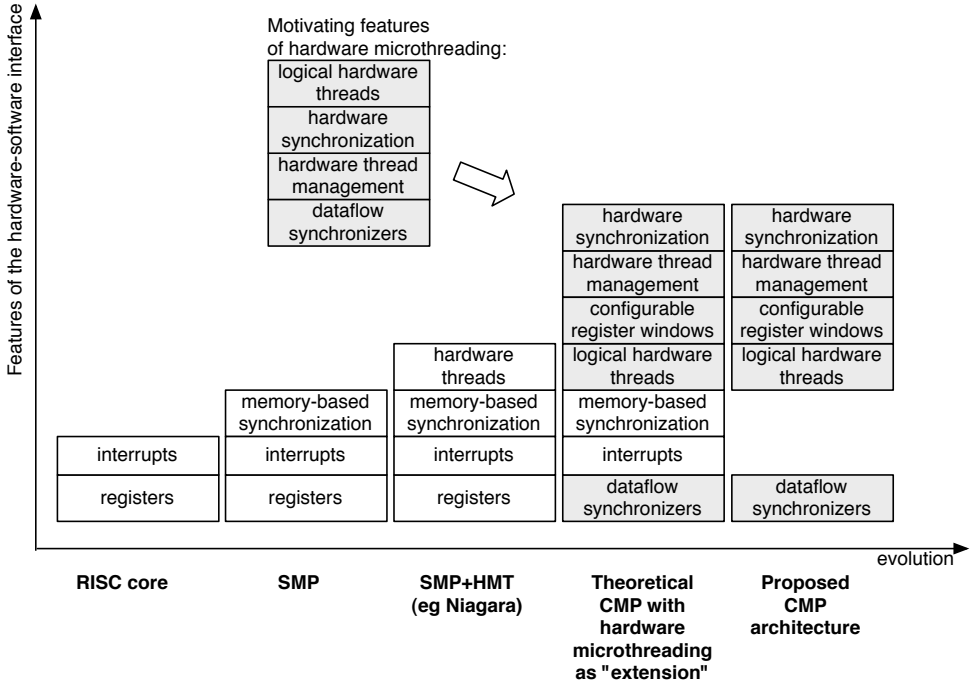


Figure 15.1: Features of the machine interface across CMP designs.

Configurable register windows, although characteristic of the design, are not motivating features; they are "merely" a quite useful optimization towards increasing utilization of large register files.

synchronization. Consequently, they decided to omit the corresponding support from their envisioned implementation of hardware microthreading, so as to simplify the hardware logic and thus increase execution efficiency further. The resulting CMP architecture, which we detailed in part I, is also depicted in fig. 15.1.

15.2 Inner vs. outer questions: a retrospective

The computer engineering steps to a product applicable to problems outside of computing science requires the participation of practitioners across multiple fields. In particular, any innovation at the level of individual components requires both to narrow down the substance of the innovation, that is what the new components look like, and to demonstrate its applicability in a larger system, that is how to integrate the new components. These are two phases of the engineering process, which we identify respectively in chapter 1 as the answers to the "inner" and "outer" question around innovation in computer architecture.

Previous work on hardware microthreading had focused mainly on answering the inner question. The publicly funded project from which this book originated, Apple-CORE, was an initiative to follow up with answers the outer question by multiple research organizations. As a member of this research community, we did not *independently* answer the inner nor the outer question around hardware microthreading; instead, we have:

1. recollected the collective work on the inner question (part I);

2. proposed enabling technology to help the collective answer to the outer question (chapters 6 and 8 to 11);
3. summarized the current collective work to the outer question (chapters 13 and 14).

The vantage point of the interface between platform provider and operating software providers provided two unique opportunities. First, we could identify several points where the answers to the outer question influence the answers to the inner question, i.e. where integration provides feedback on design. We recollect these in section 15.2.1. Then, by studying the consequences of *removing* features from the machine interface, we gained insights about what are the fundamental requirements of general-purpose computing and how they are affected by hardware microthreading. We comment on this in section 15.2.2.

15.2.1 Feedback on the inner question

By inviting operating software providers to interfere with the architecture design, we enabled a form of hardware/software co-design. For example, as we show in chapter 4, separate compilation in software engineering mandates a specific choice for the implementation of configurable register windows, which would be otherwise neutral from the architect's perspective. More generally, as we argue in chapter 5, early feedback helps saving on development costs by avoiding design points which hinder further integration.

Beyond potential benefits in optimizing the overall engineering process, answering the outer question early also reveals additional requirements on the design of individual components. For example, as we show in chapter 9, the efficient provisioning of TLS, which is a logical feature of software, in turn requires ISA extensions and special support from caches. As we discuss in chapter 14, logical mutual exclusion considered as a software service requires to extend the proposed hardware synchronization protocol and may constrain the chip topology. As we argue in chapters 9 and 14, further support for process isolation and virtual memory addressing will require mechanisms for process identification in the hardware concurrency management protocol.

These observations confirm that combining integration with design is not merely a benefit to the platform provider; it is actually a necessity to avoid the HIMCYFIO pitfall we identified in chapter 1.

15.2.2 General-purpose computing and hardware microthreading

Beyond the productivity advantages of automated computing, the advent of *general-purpose* computers was a key step forward in the progress of mankind: it democratized the process of constructing solutions in software to both current and *future* problems at virtually no cost.

Meanwhile, serious upcoming technology challenges are the cause of a flurry of contemporary research activities around multi-core architectures and parallel programming. As we acknowledged in chapters 1 and 2, these challenges can be partially addressed by creating faster and more efficient chips *tailored to specific applications*. However we argue that any durable solution should strive to preserve generality in our computing artefacts. In the context of hardware microthreading, this issue is relevant because some features that make legacy processors fully general, namely the semantics of main memory and asynchronous event delivery, are altered in the proposed CMP design.

As a first step towards illustrating the generality of the design, we demonstrated that the resulting chip can run arbitrarily sequential computations described by a general-purpose

language, by constructing a C compiler and implementing minimal support for a C library (chapter 6). Towards asynchronous event delivery, we are able to use thread creation instead of control flow interrupts for external I/O (chapters 5 and 6). We discovered ground for further research towards supporting traps and exceptions (chapter 14), which we mention again below in section 15.3; however we believe that there are no further conceptual issues with claiming that spontaneous creation of logical threads constitutes a workable substitute to interrupt delivery.

In contrast, we discovered fundamental issues related to the interaction between cores and memory when using the proposed CMP for parallel computations. Beyond the requirement that multiple logical threads must each independently have access to TLS to fully simulate communicating processes where each process is Turing-complete, a topic which we explore in chapters 9 and 12, we discovered two major concerns.

Our first concern is that the generality of a parallel computing system, as opposed to a single-processor computer, is dependent on the ability to define arbitrary communication patterns between simultaneously running processing agents. By simplifying its individual cores and weakening its memory semantics, two choices motivated by potential reductions of hardware and energy costs, the designers of the proposed CMP have abandoned the memory-based mechanisms to define arbitrary parallel computations that had been devised in previous SMP architectures. To our knowledge, no prior work had acknowledged this differentiation as we do in chapter 7. To preserve generality while keeping the benefits of a simplified design, we propose in chapter 7 new semantics for programming languages to define arbitrary communication patterns over a weakly consistent memory and a dedicated synchronization protocol, such as found in the proposed CMP. Then, in chapter 12, we show how the proposed CMP could theoretically run programs assuming previous abstract concurrent execution models.

Our second concern is that software audiences rely on the ability to abstract hardware resources in programming languages, a requirement which mandates mechanisms to *virtualize* resources. As we argue in chapter 10, each resource type potentially needs different mechanisms. For storage and I/O, a well-know mechanism is the address space virtualization provided by address translation in hardware. This facility is largely preserved with hardware microthreading, although it receives a new interface and thus mandates some extra attention in operating software (chapters 5, 9 and 14). Meanwhile, the introduction of hardware microthreading implies the apparition of three new hardware resource types that now need to be virtualized: thread execution contexts, bulk synchronizers and dataflow synchronizers. For thread execution contexts, the machine interface already proposes thread virtualization by automatically multiplexing logical threads over hardware contexts (chapters 3 and 4). However, as we highlight in chapter 14, the coarser-grain activities spawned by multiple separate software applications sharing hardware resources may in turn mandate new mechanisms to virtualize entire multi-threaded processes (as opposed to single threads), which we outline at a high level in chapters 9 and 14. For bulk synchronizers, we argue in chapter 10 that the advent of declarative concurrency in programming languages obviates the need to virtualize individual bulk-created and bulk-synchronized "families" of logical threads. We have not yet found suitable mechanisms to virtualize dataflow synchronizers, and we suggest that a solution should be proposed in future work before hardware microthreading becomes fully accepted by software communities.

15.3 Where to go from here?

There are two perspectives to suggest future work.

▶ The first is the perspective of the innovator, or the research group(s) who will follow up on the inner question: research on the substance of hardware microthreading. By exposing the innovation to current software ecosystems, we revealed that the inner question has not yet been fully answered. Namely, key architectural features and research issues must be addressed before the design becomes fully suitable for use in applications:

- the *memory architecture* must be fully defined, including its topology and address-to-bank mappings and virtual address translation mechanisms. As we discussed in chapters 7 and 9 and sections 13.7.1 and 14.3, the memory system's characteristics will impact how run-time operating software will be constructed around the platform;
- the architectural support for concurrency management must be complemented with support for *asynchronous events and traps*. Barring the reintroduction of control flow preemption, it is not yet clear how software can react to unforeseen circumstances in the run-time environment. As we highlighted in chapter 14, any programming language implementer for this platform will require a clear vision on this topic before it can be used to implement larger applications;
- proper support for *process boundaries* must be introduced in the machine interface. We have specifically highlighted in chapters 6, 9 and 14 that resource reclamation (memory, cores, thread contexts) upon process termination requires the identification of resources that belong to different processes. Process boundaries are also needed to provide isolation and accountability, which we did not cover in our work but are expected from all software ecosystems. The current hardware interface does not yet address these aspects;
- proper support for *resource virtualization* must be proposed and documented. While the machine interface proposes automatic virtualization of tasks as logical threads (chapters 3 and 4) and we could propose mechanisms to virtualize bulk synchronizers (chapter 10), no mechanism yet exists to fully virtualize the dataflow synchronizers and entire processes;
- *performance and resource models* must be devised that establish a relationship between the architectural parameters and the behavior of programs. As we illustrated in chapter 13, naively implemented algorithms are able to obtain a comparatively higher throughput per unit of area than with legacy chips. However, performance scalability is actually constrained by a combination of parameters [GBK+09]: on-chip communication bandwidths, number of thread contexts per core, number of cores per L2 cache, etc. As we suggest in chapters 11 and 13, the implementation of space schedulers in operating software by third parties will only be possible once models can successfully predict the performance of program components running on different regions of the chip.

▷ The other perspective is that of the computing science community who will follow up on the outer question: developing the context where hardware microthreading can be applied. We foresee the following necessary steps:

- while the research topics listed above are being studied, *test and simulation platforms* must be developed where the updated chip designs can be evaluated across a larger range of applications, including larger workloads. During our own work, we had access only to a low-level simulation of the multi-core design running at a few million

instructions per second and a single-core implementation on an FPGA. To our knowledge, this is a common limitation in most contemporary research projects in computer architecture. To support larger workloads during testing, including industry-standard benchmarks, *faster simulators* and emulators for many-core chips must be developed: either higher-level simulations in software running on distributed computational clusters, or by connecting multiple FPGA platforms via high-speed networks that mimic on-chip topologies;

- to support the diversity of existing software frameworks, further *operating software* must be developed to expose the generality we revealed in chapter 12. We could envision, for example, custom implementations of the POSIX threading library or ISO C11's concurrency management primitives, an implementation of MPI, a back-end for an OpenCL compiler, or support in the run-time system of fashionable productivity languages like UPC or Chapel;
- an *exploration of the architecture design space* must take place to determine a balanced configuration of cache sizes, number of thread contexts per core, number of cores, number of synchronizers per core, and NoC characteristics for a given technology and area budget. This is necessary to eventually lay out the floorplan of a chip to estimate its power requirements prior to manufacturing.

We highlight that these steps are not specific to the design we presented in part I. They apply to any innovation in general-purpose microprocessor design that pushes for large-scale parallelism on chip.

Chapter 16

Epilogue on the outer question

Hardware microthreading may be a suitable answer to the obstacles currently standing in the way to better, faster and more efficient building blocks for general-purpose computers. Yet the evaluation of this innovation has been carried out so far only via small computation kernels performing no I/O, and thus far removed from contemporary applications and software ecosystems.

As we have illustrated, innovation in computer architecture, while necessary, is not a task that can be carried out in isolation. Practitioners must associate with their peers in software engineering and systems engineering to define *platforms* that contextualize the individual components and bring them to the scrutiny of audiences external to the field. We have argued that this contextualization, which we name the *outer question*, cannot be carried out by one architect, or even a small group. Individual, human creators have separate personal interests and fields of expertise; the diversity of expertise needed to create entire computing platforms exceeds the management and organizational capacity of a small research group.

Historically, innovation in computer architecture occurred first in public organizations funded by governmental and corporate organizations. As computers became commercial products and their audiences widened, innovation was then carried out in partnership between academic and corporate organizations, the latter being responsible for carrying the financial and human effort necessary for manufacturing and commercialization. This transition occurred across all uses of computers, from small embedded systems to datacenters. Meanwhile, the definition of *general-purpose* computing platforms was instrumental to power the IT industry, in particular the industry of software engineering, and decouple the production of software from the production of hardware.

Since the turn of the 21st century, powerful market forces are at play to redefine the place of general-purpose computers. Before the last decade, users could define the function of their computing devices separately from form. They could do so by acquiring software, but also by *writing their own software*, separately from the acquisition of the platform. In contrast, the last decade has then seen the advent of smart terminals and entertainment platforms, whose form and function are bundled by the manufacturer and not separable by the user. These devices have displaced commodity general-purpose platforms. They progressively erase the incentive to educate non-technical audiences about the benefits of carrying out their own innovation in the software domain.

On December 27th, 2011, famous technology activist Cory Doctorow addressed the 28th Chaos Communication Congress in Berlin with his keynote titled *The coming war on general computation: the copyright war was just the beginning*. He summarized the essence of his message thus:

> *The problem is twofold: first, there is no known general-purpose computer that can execute all the programs we can think of except the naughty ones; second, general-purpose computers have replaced every other device in our world. There are no airplanes, only computers that fly. There are no cars, only computers we sit in. [...] Consequently anything you do to "secure" anything with a computer in it ends up undermining the capabilities and security of every other corner of modern human society.*
>
> *And general purpose computers can cause harm—whether it's printing out AR15 components, causing mid-air collisions, or snarling traffic. So the number of parties with legitimate grievances against computers are going to continue to multiply, as will the cries to regulate PCs.*
>
> *The primary regulatory impulse is [...] to create computers that run programs that users can't inspect or terminate, that run without users' consent or knowledge, and that run even when users don't want them to.*
>
> *The upshot: a world [...] where everything we do to make things better only makes it worse, where the tools of liberation become tools of oppression.*
>
> *Our duty and challenge is to devise systems for mitigating the harm of general purpose computing without recourse to spyware, first to keep ourselves safe, and second to keep computers safe from the regulatory impulse.*

On the one hand, we could simply acknowledge the evolution denounced by Doctorow as a natural effect of a free market where consumers have decided, through their purchasing power, their preference for pre-programmed functions and corporate control over features. On the other hand, we could worry about an opportunity loss, that of educating a larger number of individuals to the power of generality. This loss is especially relevant today. The diversity of types of basic building blocks found in commercial computing systems since the late 1980's has barely changed: beyond pipelines and registers (1950's), caches (1960's), virtual memory (1970's), and RISC and programmable logic (1980's), later computers have merely become different combinations of these blocks, with upgraded semantics (e.g. out-of-order execution); meanwhile, the entire industry is currently facing fundamental technology challenges. It seems to us that a large diversity of creative approaches by individuals empowered to innovate will be needed to overcome these challenges. Also, to guarantee equal opportunity and chance in the dynamics of innovation, it seems to us unwise to rely on private corporations and let them capture innovative types behind closed doors. An alternate route seems tractable today, because the advent of mass-manufactured generic FPGA substrates makes it affordable for individuals and small research groups to innovate in computer architecture and demonstrate their inventions at a low cost.

These considerations are, of course, an essentially political comment about the human dynamics around computing science. Yet we must acknowledge these dynamics to make informed technological and scientific choices around efforts towards innovation.

Coming back to hardware microthreading, the "economically correct" route forward in the current landscape would be to partner with a corporate entity, select or devise an application domain and shape a product to be marketed in that domain. A prerequisite to that route, however, is often to let the corporation own the invention and prevent third parties from reproducing and altering its design. Even if such exclusive licensing is not demanded, investing effort towards a product captures the innovative energy in the direction of the specific application domain that the product is targeting. The alternative is transparency and

openness, i.e. working towards maximum exposure and opening the opportunity for multiple organizations to devise derived platforms independently. The trade-off is about wealth and fame: investing in a product incurs either short-term wealth and fame or short-term destitution, whereas investing in a campaign for transparency incurs neither.

We have answered in our dissertation the outer question around hardware microthreading by exploring the route towards transparency; our answer is to be found in the journey, not the end result. On this path, we have built a programming environment upon the C language and the Unix operating system, which are both technologies currently in shared control of communities from both for-profit and non-profit organizations. We have motivated this choice by showing that the perspective of these communities can reveal early shortcomings in the innovation, which a product-based approach would only reveal after a larger investment. In our design choices, which we exposed, we prioritized generality over efficiency or specialization to a specific domain. We have publicly detailed the inner workings of the resulting platform, so that third parties can reproduce our findings and develop their own software tools anew if so desired. We have illustrated the applicability of our technology by showing that it was used successfully by third parties, in particular to show the higher throughput per unit of area and time of the proposed design compared to contemporary hardware. The output of our work, reported on in this book, is mere information: technical and scientific insight, plus general-purpose operating software and programming tools to equip an implementation of hardware microthreading compatible with the interface presented in chapter 4. Our contributions, including this book itself, can be further reproduced and studied free of charge without the explicit agreement of their author.

Through the looking glass

As parting words, we recognize that the history of computing has not recently seen instances where innovations in computer architecture were brought to the hands of the general public without heavy involvement and investment from large corporations. It may well be that the various steps we identified in chapter 15 will be a show stopper for the specific research direction considered, because the proposed concepts diverge too much from industry-established standards. Or, more likely, some architectural concepts will be partially retrofitted by existing industry leaders in legacy designs, without further consideration for our proposed platform. Meanwhile, we cannot ignore either the democratization of architecture research enabled by FPGA technology, or the realizability of cheap platforms for education like Raspberry Pi[1]. One can only barely imagine the potential of outsourcing the various issues identified previously to communities of hobbyists and younger/amateur scientists equipped with low-cost FPGA boards, produced and distributed in the manner of Raspberry Pi, and exchanging design directions publicly with the help of the Internet. This conjunction of opportunities is recent, and may well be the vehicle sufficient to overcome the traditional threshold to innovation in architecture research.

[1] `http://www.raspberrypi.org/`: this UK-based, non-profit foundation, whose primarily interest is education, brought a fully-programmable, state-of-the-art general-purpose computer equipped uniquely with F/OSS to the general public for the price of a couple meals (US$25). Their first production batch was sold out before the first shipping, and their pre-sales at the start of 2012 have so far exceed prior estimates.

Part IV

Appendices

Appendix A

Information sources for the hardware interface

The description work in chapters 3 and 4 was bootrapped by a preliminary analysis of what information was already available. The following is an exhaustive list of the sources related to the hardware design at that point (late 2008):

‹i› a library of 24 short example programs written in assembly source, listed in table A.1;
‹ii› 17 academic publications containing descriptions of the hardware architecture, listed in table A.2;
‹iii› an internal (unpublished) technical report describing the machine instructions that control multithreading [LB08];
‹iv› program source code for "`mtsim`," a functional emulator of an abstract multi-core system using the proposed architecture, using assembly source as input;
‹v› program source code for "`mtaxpasm`," a translator from assembly source to machine code suitable for use with `mtsim`,
‹vi› program source code for "MGSim," a cycle-accurate component-level simulation of a multi-core system using the proposed architecture, using binary machine code as input;
‹vii› program source code for a modified version of the GNU assembler and linker[1], which superseded `mtaxpasm` to target MGSim;
‹viii› the author of MGSim, the author of `mtsim`, and the other research staff in charge of the architecture design.

When faced with conflicting information, we investigated the MGSim machine emulator at that time, advertised as the "reference implementation," to resolve the inconsistencies. This ensures our description is consistent with [Lan07, Lan1x]. Otherwise, the information in sections 3.2 to 3.4 was obtained from sources ‹ii›, ‹vi› and ‹viii›. The information in chapter 4 was obtained from sources ‹i› to ‹iii› and ‹vi› to ‹viii›.

[1] `http://www.gnu.org/software/binutils/`

Program	Description
`fft/ fft_seq_u`	unoptimized sequential 1D FFT kernel
`fft/ fft_seq_o`	" hand-optimized
`fft/ fft_mt_u`	" unoptimized, multithreaded
`fft/ fft_mt_o`	" hand-optimized
`fibo/ fibo`	multithreaded program that computes the Nth element of the Fibonacci sequence
`livermore/ l1_hydro`	multithreaded hydro fragment from the Livermore suite [McM86] (integers only)
`livermore/ l2_iccq`	multithreaded incomplete cholesky conjugate gradient (integers only)
`livermore/ l3_innerprod`	multithreaded vector-vector product (integers only)
`livermore/ l4_bandedlineareq`	multithreaded banded linear equations kernel (integers only)
`livermore/ l5_tridiagelim`	multithreaded tri-diagonal elimination below diagonal (integers only)
`livermore/ l6_genlinreceq`	multithreaded general linear recurrence equations (integers only)
`matmul/ matmul0`	square matrix-matrix multiplication, integers only, sequential
`matmul/ matmul1`	" multithreaded, one level of threads
`matmul/ matmul2`	" multithreaded, two levels of threads
`matmul/ matmul3`	" multithreaded, three levels of threads
`sine/ sine_seq_u`	unoptimized, sequential sine function using 9-level Taylor expansion
`sine/ sine_seq_o`	" hand-optimized
`sine/ sine_mt_u`	" unoptimized, multithreaded
`sine/ sine_mt_o`	" hand-optimized
`sac/ sac_flat`	sequential test program containing an heterogeneous floating-point array computation, hand-compiled from a higher-level functional language
`sac/ sac_nested_local`	" multithreaded, using one core only
`sac/ sac_nested_group`	" using multiple cores

Table A.1: Example programs from the architecture test suite, December 2008

Key	Title
[BJM96]	Dynamic scheduling in RISC architectures.
[JL00]	Micro-threading: a new approach to future RISC.
[Jes01]	Implementing an efficient vector instruction set in a chip multi-processor using micro-threaded pipelines.
[LJ02]	Performance of a micro-threaded pipeline.
[Jes03]	Multi-threaded microprocessors – evolution or revolution.
[Jes04]	Scalable instruction-level parallelism.
[Jes05]	Microgrids – the exploitation of massive on-chip concurrency.
[BJ05]	The challenges of massive on-chip concurrency.
[BHJ06b]	Instruction level parallelism through mi- crothreading – scalable approach to chip multiprocessors.
[BBG+06]	A microthreaded architecture and its compiler.
[Jes06b]	Microthreading, a model for distributed instruction-level concurrency.
[BJK07]	Strategies for compiling μTC to novel chip multiprocessors.
[ZJ07]	On-Chip COMA Cache-Coherence Protocol for Microgrids of Microthreaded Cores.
[Lan07]	Developing a Reference Implementation for a Microgrid of Microthreaded Microprocessors.
[Jes08a]	Operating systems in silicon and the dynamic management of resources in many-core chips.
[Jes08b]	A model for the design and programming of multi-cores.
[BBG+08]	A general model of concurrency and its implementation as many-core dynamic RISC processors.

Table A.2: Academic publications explaining the architecture, December 2008

Appendix B

Optimal control word size analysis

Number of instructions per control word	Size of control word	Minimum I-cache line size (non-aligned)	Minimum I-cache line size (aligned)	Unused bytes in line
1	2 bits	5 bytes	8 bytes	3
2	4 bits	9 bytes	16 bytes	7
3	6 bits	13 bytes	16 bytes	3
4	8 bits	17 bytes	32 bytes	15
5	10 bits	22 bytes	32 bytes	10
6	12 bits	26 bytes	32 bytes	6
7	14 bits	30 bytes	32 bytes	2
8	16 bits	34 bytes	64 bytes	30
9	18 bits	39 bytes	64 bytes	25
10	20 bits	43 bytes	64 bytes	21
11	22 bits	47 bytes	64 bytes	17
12	24 bits	51 bytes	64 bytes	13
13	26 bits	56 bytes	64 bytes	8
14	28 bits	60 bytes	64 bytes	4
15	30 bits	64 bytes	64 bytes	0

Table B.1: Impact of control bits on I-cache line utilization.

Assuming 32-bit instruction formats.

Instruction width	Number of instructions per control word	Minimum I-cache line size (aligned)
8 bits	1	2 bytes
8 bits	3	4 bytes
8 bits	6	8 bytes
8 bits	25	32 bytes
8 bits	51	64 bytes
8 bits	102	128 bytes
16 bits	7	16 bytes
16 bits	14	32 bytes
16 bits	455	1024 bytes
24 bits	1	4 bytes
24 bits	315	1024 bytes
32 bits	15	64 bytes
48 bits	5	32 bytes
64 bits	31	256 bytes
64 bits	62	512 bytes

Table B.2: Number of instructions per control word that maximize I-line utilization.

Parameters suitable for minimal I-cache line sizes lower than 1KiB.

Appendix C

Running example with machine code

This appendix illustrates one of the possible machine interfaces described in chapter 4 by looking at a working example.

The first part of the example is given in table C.1. We consider the *structure* of the memory image first. The first bits are control bits (sections 3.2.1 and 4.4 and Appendix B). These are defined 32 at a time, in groups of 2 bits per following instruction. In this example, the hexadecimal value 0x00030400 can be decomposed from LSB to MSB as 0, 0, 0, 0, 0, 1, 0, 0, 3 followed by 7 groups with value 0. The first value is ignored as it refers to the control word itself; the next 8 values (0, 0, 0, 0, 1, 0, 0, 3) are the control bits for the 8 instructions that follow, with values 0 for all instructions except "setlimit" and "mov" which have values 1 and 3, respectively. The value 1 indicates to the pipeline's fetch unit to switch to a different thread, whereas the value 2 indicates to terminate the thread execution; both values can be combined using a binary OR. Since the "mov" instruction terminates the thread, there are no more instructions afterwards in this thread program. The memory image thus contains

Offset	Machine code (hex)				Textual representation
0x00	00	04	03	00	(control 0x00030400)
0x04	00	00	60	08	allocate $3
0x08	00	14	e0	47	mov 0, $0
0x0c	01	34	e0	47	mov 1, $1
0x10	22	51	40	40	subl $2, 2, $2
0x14	5f	00	62	04	setlimit $3, $2
0x18	00	00	60	18	setregs $3, 0, 0, 0, 0
0x1c	0b	00	60	10	cred $3, 0x48
0x20	1f	04	e3	47	mov $3, $31
0x24	1f	04	ff	47	(padding)
0x28	00	00	fe	2f	(padding)
0x2c	1f	04	ff	47	(padding)
0x30	00	00	fe	2f	(padding)
0x34	1f	04	ff	47	(padding)
0x38	00	00	fe	2f	(padding)
0x3c	1f	04	ff	47	(padding)

Table C.1: Memory image for the program "`fibo`", late 2008

padding at this point to complete the length of the cache line: the fetch unit loads entire cache lines at a time (64 bytes here) in the instruction cache.

We then consider the *semantics* of the memory image. The system initially starts a single thread. The initial program counter is configurable externally to the system (either as a hardware parameter or a pointer in a memory image), and without further specification it default to 0. This seems inconsistent with the program image, because the data at address 0 is the control word. In fact, the fetch unit always automatically skips the first 4 bytes of a cache line when fetching instructions. This implies that the first instruction is really at address 4, in this case "allocate." Here we may wonder why no synchronizer configuration is present here before the start of the program, as required by section 4.3.3. In fact, there is a special exception for the initial thread: the hardware does not require this synchronizer specification for the first thread and always allocates 31 local (private) synchronizers.

As per section 4.3.1, the "allocate" instruction is the first step of a family creation, and reserves a yet unused *bulk creation context* from a dedicated memory component on chip. It then stores the address of this context into the target operand of the instruction—here the synchronizer with the name "$3."

The next two "mov" instructions load the constants 0 and 1 in synchronizers $0 and $1, respectively.

The next "subl" instruction substracts 2 from the value of synchronizer $2, and stores the result into $2. This seems nonsensical, since synchronizer $2 was not set until this point. This is actually a feature of the software emulation platform MGSim: individual processor synchronizers can be set to predefined values before the system is activated. This is a crude way to provide external input to the program[1]. All synchronizers that are not predefined this way are initially in the "empty" state. We can confirm this by running the program without input: execution stops at the "subl" instruction, waiting for input which is never defined. If the synchronizer is predefined, execution flows past the instruction.

The next instruction "setlimit" configures (section 4.3.2.3) the limit parameter of the family context reserved by "allocate," using the result of "subl." However, we know that this instruction is also marked with the special control bits that indicate the fetch unit to switch to a different thread. This seems nonsensical, as we have seen that initially only one thread is created. Actually, the fetch unit has a special case for the situation where only one thread is active: in that case, the "switch" annotation is ignored and the next instruction is fetched from the same thread. From this point it may become unclear why the "switch" annotation was specified at all. From the same sources we learn that this was done for consistency: *the switch annotation should placed on all instructions that may suspend a thread*, to ensure that the next slots in the pipeline are always occupied with work if there are other threads active.

The next instruction "setregs" configures the offset, in the creating thread's visible window of synchronizers, of the channel endpoints with the created family. Here we recognize an interface with "hanging" synchronizers (section 4.3.3.3). Here all offsets are set to 0, which means that the first synchronizer name used to communicate will be "$0" for each channel. The endpoint synchronizer names are chosen contiguously from the first. This instruction clarifies the role of the two preceding "mov" instructions: by placing values in synchronizers "$0" and "$1," they have prepopulated the input side of the communication with the created family. Interestingly, the instruction only specifies one offset for each of the

[1] While this is clearly an unrealistic feature for a hardware implementation, it is a simple and convenient system that automates execution of the same program for multiple input values.

channel categories (integer "global," integer "shared," floating-point "global," floating-point "shared"); meanwhile, we know from section 4.3.3.3 that the architecture connects the outgoing "shared" channels of the last thread to the parent thread. In fact, the 2nd and 4th parameter to "setregs" specify *both* the endpoint of the outgoing value to the first thread, and the incoming value from the last thread. Each of these offset specifications may be eventually ignored by the hardware if no channel of the corresponding type is defined.

The next instruction "cred" creates the family of threads using the family context identified by its 1st operand (here $3, set to the result of "allocate"), and the program counter identified by its 2nd operand (here 0x48). This is a "fused creation" (section 4.3.1.2): the execution of "cred" further binds the 1st operand to a *future* termination value for the created family. This means that the named synchronizer (here $3) is set to the "empty" state, with the contract that the hardware will change it to the "full" state when all threads in the family terminate. This implies that any further instruction that uses the same synchronizer as input operand will cause the thread to suspend until the family terminates.

Note that both "crei" and "cred" exist; the former accepts a synchronizer operand and an offset for the program counter ("create Indirect"), while the latter accepts an immediate value ("create Direct"). They have otherwise the same semantics, and are collectively named "create."

The final instruction "mov" reads from the named synchronizer $3; as seen above this causes the thread to wait until the created family terminates. As per section 4.3.1.2, with a fused creation style, the context reserved by "allocate" is automatically released upon termination and thus made available for subsequent allocations by the same thread or other threads on the processor.

Notwithstanding the question of the validity of address 0x48 in the "cred" instruction, discussed below, we can summarize the following:

- upon system start-up, execution starts with 1 thread using 31 local synchronizers and a configurable initial program counter;
- the synchronizer configuration is omitted for the program of the initial thread;
- the fetch stage skips the first 4 bytes of each cache line;
- control words are organized in groups of 2 bits, one per instruction; value 2 indicates "end thread", value 1 indicates "switch;", and the binary OR of both values is possible;
- switch does not really occur in the fetch unit if there is only 1 thread active;
- initially all synchronizers have the state "empty," except in MGSim where they can be predefined to input values by the user;
- programmers should annotate all instructions that may suspend a thread with the "switch" control bits;
- "cred" and "crei" cause the actual creation of threads for a family; their output in the creating thread is a future on the family's termination, and the target synchronizer operand of both instructions is "tied up" to the execution of the family until the family terminates (and thus cannot be reused until then).
- "allocate" reserves a family context from a finite-size hardware data structure, used subsequently by "cred" and "crei" and released for further reuses when the family terminates;
- "setlimit" configures the limit parameter for the named family identified by its first input operand, using the value of the 2nd input operand; other examples show that the instructions "setstart," "setstep" and "setblock" also exist which configure resp. the start, step and "block" parameters for the family (cf. sections 4.3.2.2 and 4.3.2.3);

Offset	Machine code (hex)				Textual representation
0x40	d0	00	00	00	(control 0x000000d0)
0x44	40	00	00	00	(reg. spec. 0x00000040)
0x48	01	00	43	40	addl $2, $3, $1
0x4c	00	04	e3	47	mov $3, $0
0x50	1f	04	ff	47	(padding)
0x54	00	00	fe	2f	(padding)
0x58	1f	04	ff	47	(padding)
0x5c	00	00	fe	2f	(padding)
0x60	1f	04	ff	47	(padding)
0x64	00	00	fe	2f	(padding)
0x68	1f	04	ff	47	(padding)
0x6c	00	00	fe	2f	(padding)
0x70	1f	04	ff	47	(padding)
0x74	00	00	fe	2f	(padding)
0x78	1f	04	ff	47	(padding)
0x7c	00	00	fe	2f	(padding)

Table C.2: Continuation of table C.1

- "setregs" configures the channel endpoints in the creating thread for the named family identified by its input operand, using 4 constant values that denote offsets in the thread's virtual synchronizer window. The 4 offsets specify the integer "global," integer "shared," floating-point "global" and floating-point "shared," in this order; the 2nd and 4th offsets are used both for the outgoing endpoint to the first thread and the incoming endpoint from the last thread.

We can then proceed with the study of the rest of the memory image of the example, listed in table C.2. As expected, there is data in memory at the address 0x48, specified by the "cred" instruction at address 0x18. As previously, we can decompose the structure of this cache line first. The control bits indicate values 0, 0, 1, 3, followed by 12 2-bit blocks with value 0. The first value corresponds to the control word itself, which leaves values 0, 1, 3 for the next instructions in the cache line. Value 1 (switch) applies to the "addl" instruction, whereas 3 (end thread) applies to "mov." This indicates that "mov" is the last instruction; the rest of the cache line is filled with padding.

As to the semantics, the "cred" instruction indicates that the start of the thread program lies at address 0x48. The synchronizer window configuration immediately precedes, here at address 0x44. This value should be decomposed first in blocks of 16 bits from LSB to MSB, here 0x0040 and 0x0000. The 1st block corresponds to integer synchronizers, the 2nd to floating-point synchronizer. Each block is then further decomposed from LSB to MSB in groups of three 5-bit values, here thus 0, 2, 0 for integers and 0, 0, 0 for floats. In each block, the first value indicates the number of global channels, the 2nd value indicates the number of shared channels, and the 3rd value indicates the number of local (private) synchronizers. In this example the configuration defines only 2 shared channels, i.e. 4 synchronizer endpoints as per section 4.3.3.3 (two incoming, two outgoing).

The first instruction in the thread program is then "addl," with inputs $2 and $3 and output $1. As this interface maps windows in the order G-S-L-D (section 4.3.3.4), the outging channel endpoints are mapped before the incoming endpoints. Since there are no

other synchronizers defined in this example, the names \$0 and \$1 thus point to the outgoing channels, and \$2 and \$3 point to the incoming channels. Therefore, the "addl" instruction can be understood to read from each incoming channel, and produce its output on the 2nd outgoing channel.

The subsequent "mov" instruction then propagates the value from the 2nd incoming channel to the 1st outgoing channel, and the thread then terminates.

Finally, we need to look again at the previous fragment from table C.1 to understand where the endpoints of the first and last links are connected (the outgoing part of the incoming channels of the first thread in the family, and the incoming part of the outgoing channels of the last thread). As described above, they are specified by "allocate," in this case all endpoints start from the first local synchronizer, here \$0. Therefore, the two endpoints of the first shared channel are mapped onto synchronizer \$0 of the creating thread, and the 2nd shared channel endpoints are mapped onto \$1. It is relevant to observe at this point that these two synchronizers are initialized to 0 and 1 by the creating thread.

From a slightly higher level perspective, we can describe the expected behavior of this program as follows: the program takes an input value, let us call it N. It then creates a family of $N-2$ threads linked using two daisy-chains of synchronous channels; and initializes the two inputs of the first thread to 0 an 1, respectively. Finally, it waits on termination of the family. Meanwhile, each created thread sums its two inputs into the 2nd output channel, and propagates the 2nd input to the 1st output channel. Inductively, the values produced by the last threads are the Nth and $N-1$th values of the Fibonacci sequence. It is then possible to observe externally the final state of synchronizers \$0 and \$1 in the creating thread after it terminates to obtain the program's result.

We can summarize further:

- the first intruction of a thread program cannot be placed directly after the control word, so as to leave room for the synchronizer configuration information which must immediately precede;
- the synchronizer configuration word, placed immediately before the first thread program instruction, is specified as two 16-bit blocks containing 3 5-bit values each, defining the number of global, shared, local synchronizers for integers and floats, in this order;

We can see an example of synchronizer sharing in fig. C.1. This state was obtained after running a thread program with synchronizer configuration 4, 6, 9, 0, 0, 0, which in turn created a family of 3 threads whose synchronizer configuration was 2, 3, 4, 0, 0, 0. The "setregs" operation indicated offsets 1, 5, 0, 0 for the channel endpoints in the creating thread. The shaded areas indicate the resulting mapping of physical synchronizers to logical names in the visible synchronizer windows of each thread; there are 6 unused synchronizer names in the creating thread, and 19 unused synchronizer names in each of the 3 threads in the created family.

In contrast, if the implementation was using separate synchronizers (section 4.3.3.3), as opposed to "hanging" in the previous case, the same configuration would yield the pattern illustrated in fig. C.2.

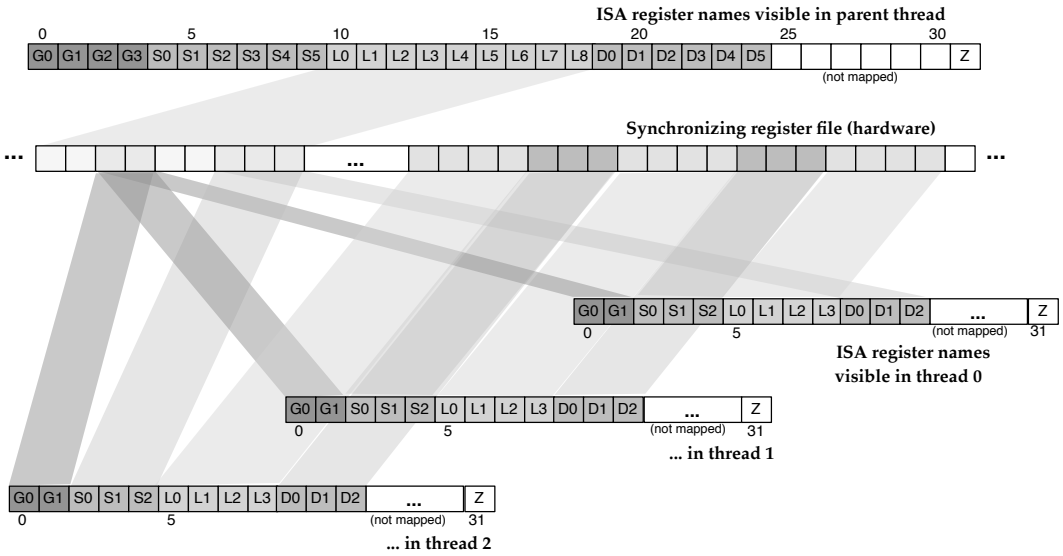


Figure C.1: Observed synchronizer sharing between a creating thread and a family of 3 threads.

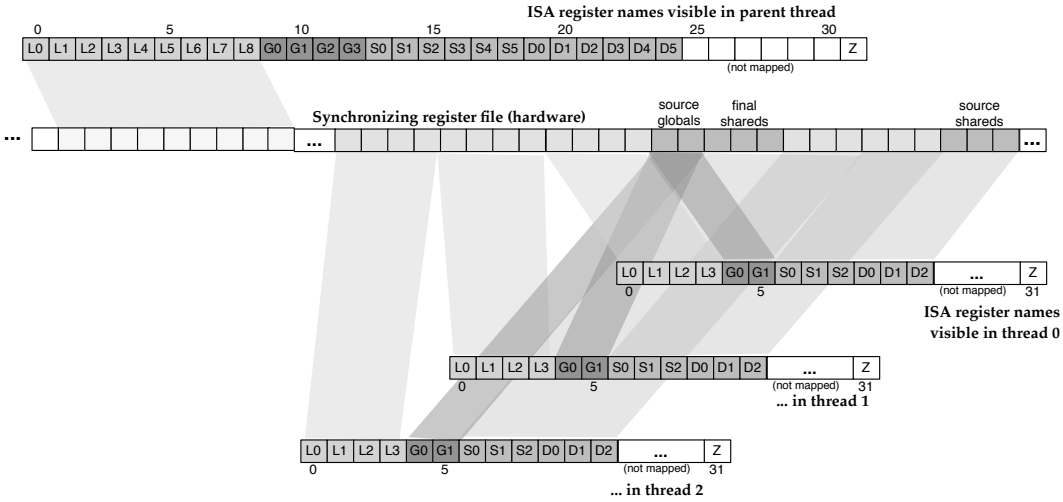


Figure C.2: Observed register sharing between a creating thread and a family of 3 threads.

Appendix D

Assembly instruction formats and machine encoding

Mnemonic	Description	Sync. / Async.	Asynchronous completion condition
`crei`, `cred`	Start bulk creation of logical threads	Async.	Bulk creation has started.
`sync`	Request bulk synchronization.	Async.	All bulk created threads have terminated.
`create`	Start bulk creation and request bulk synchronization.	Async.	All bulk created threads have terminated.◇
`creba`, `creba/s`, `crebas`	Issue a combined allocation, configuration and creation request with parameters stored in memory at the specified location.	Async.	Bulk creation has started†
`crebi`, `crebi/s`, `crebis`	Issue a combined allocation, configuration and creation request with parameters stored in memory, looked up by index in a table in memory starting at an address stored in a predefined ancillary system register on the local core.	Async.	Bulk creation has started†
`break`	Stop creating new logical threads in the current bulk creation, but let existing logical threads to completion.	Sync.	N/A

◇ The "`create`" instruction on UTLEON3 implements fused creation as described in section 4.3.1.2.
† Only the "`*s`" variants acknowledge bulk creation. "`creba`" and "`crebi`" do not output a future.

Table D.1: Description of the family control instructions.

Mnemonic	Description	Sync. / Async.	Asynchronous completion condition
`putg, fputg, puts, fputs`	Write a remote register/synchronizer.	Async.	None: fully asynchronous.
`getg, fgetg, gets, fgets`	Read a remote register/synchronizer.	Async.	Remote value received.‡

‡ The "`get`" instructions to not suspend if the remote synchronizer is not *full* (no dataflow synchronization across cores); instead an undefined value is returned.

Table D.2: Description of the register-to-register communication instructions.

Mnemonic	Description	Sync. / Async.	Asynchronous completion condition
`allocate, allocates, allocate/s, allocatex, allocate/x`	Allocate a bulk creation context.	Async.	Remote context has been allocated.⋆
`release`	Release a previously allocated context	Async.	None: fully asynchronous.
`setstart, setlimit, setstep, setblock`	Configure the bulk creation parameters after allocation and prior to creation.	Async.	None: fully asynchronous.

⋆ The "`*s`" and "`*x`" variants wait until a context becomes available. The base variant returns a special value to signal if no context was available.

Table D.3: Description of the bulk context management instructions.

Mnemonic	Description
`ldbp`	Load base pointer: output the base TLS address (cf. chapter 9).
`ldfp`	Load end pointer: output one address past the end of the TLS space.
`gettid`	Output the local address of the thread context (within its core).
`getfid`	Output the local address of the bulk synchronizer (within its core).
`getpid`	Output the address of the current placement (cf. chapter 11).
`getcid`	Output the address of the local core.
`getasr`	Read the content of an ancillary system register on the local core.

Table D.4: Description of miscellaneous instructions.

These instructions retrieve local state relative to the logical thread that issues them.

Assembly format		Encoding							
Pattern	Overlaps/Extends	op1	rd	op3	rs1	i	$op_{\mu T}$	$ASI_{\mu T}$	rs2
								imm9	
`allocate %rd`	`rdasr %asr20, %rd`	10	rd	101000	0x14	0	0x1	-	00000
`create %rs2, %rd`	`rdasr %asr20, %rd`	10	rd	101000	0x14	0	0x2	-	rs2
`gettid %rd`	`rdasr %asr20, %rd`	10	rd	101000	0x14	0	0x3	-	00000
`getfid %rd`	`rdasr %asr20, %rd`	10	rd	101000	0x14	0	0x4	-	00000
`launch %rs2`	`wrasr %rs1, %asr20`	10	0x14	110000	rs1	0	0x1	-	00000
`setstart %rs1, %rs2`	`wrasr %rs1, %asr20`	10	0x14	110000	rs1	0	0x2	-	rs2
`setstart %rs1, imm9`	`wrasr %rs1, %asr20`	10	0x14	110000	rs1	1	0x2	imm9	
`setlimit %rs1, %rs2`	`wrasr %rs1, %asr20`	10	0x14	110000	rs1	0	0x3	-	rs2
`setlimit %rs1, imm9`	`wrasr %rs1, %asr20`	10	0x14	110000	rs1	1	0x3	imm9	
`setstep %rs1, %rs2`	`wrasr %rs1, %asr20`	10	0x14	110000	rs1	0	0x4	-	rs2
`setstep %rs1, imm9`	`wrasr %rs1, %asr20`	10	0x14	110000	rs1	1	0x4	imm9	
`setblock %rs1, %rs2`	`wrasr %rs1, %asr20`	10	0x14	110000	rs1	0	0x5	-	rs2
`setblock %rs1, imm9`	`wrasr %rs1, %asr20`	10	0x14	110000	rs1	1	0x5	imm9	
`setthread %rs1, %rs2`	`wrasr %rs1, %asr20`	10	0x14	110000	rs1	0	0x6	-	rs2
`setthread %rs1, imm9`	`wrasr %rs1, %asr20`	10	0x14	110000	rs1	1	0x6	imm9	
`break`	`wrasr %g0, %asr20`	10	0x14	110000	00000	0	0xA	-	00000

Table D.5: MT instruction set extensions implemented in UTLEON3.

Non-prefixed values are in base 2, and values prefixed with "0x" in base 16.

Assembly format		Encoding							
Pattern	Overlaps/Extends	op1	rd	op3	rs1	i	$op_{\mu T}$	$ASI_{\mu T}$	rs2
								imm9	
`allocate %rd`	`rdasr %asr20, %rd`	10	rd	101000	0x14	0	0x1	-	00000
`allocate %rs2, %rd`	`rdasr %asr20, %rd`	10	rd	101000	0x14	0	0x1	-	rs2
`allocate imm9, %rd`	`rdasr %asr20, %rd`	10	rd	101000	0x14	1	0x1	imm9	
(reserved UTLEON3)	`rdasr %asr20, %rd`	10	rd	101000	0x14	0	0x2	-	rs2
`gettid %rd`	`rdasr %asr20, %rd`	10	rd	101000	0x14	0	0x3	-	00000
`getfid %rd`	`rdasr %asr20, %rd`	10	rd	101000	0x14	0	0x4	-	00000
`getpid %rd`	`rdasr %asr20, %rd`	10	rd	101000	0x14	0	0x5	-	00000
`getcid %rd`	`rdasr %asr20, %rd`	10	rd	101000	0x14	0	0x6	-	00000
`crei %rs2, %rs1`	`rdasr %asr20, %rs1`	10	rs1	101000	0x14	0	0x7	-	rs2
`cred imm9, %rs1`	`rdasr %asr20, %rs1`	10	rs1	101000	0x14	1	0x7	imm9	
`sync %rs2, %rd`	`rdasr %asr20, %rd`	10	rd	101000	0x14	0	0x8	-	rs2
`allocates %rd`	`rdasr %asr20, %rd`	10	rd	101000	0x14	0	0x9	-	00000
`allocates %rs2, %rd`	`rdasr %asr20, %rd`	10	rd	101000	0x14	0	0x9	-	rs2
`allocates imm9, %rd`	`rdasr %asr20, %rd`	10	rd	101000	0x14	1	0x9	imm9	
`allocatex %rd`	`rdasr %asr20, %rd`	10	rd	101000	0x14	0	0xA	-	00000
`allocatex %rs2, %rd`	`rdasr %asr20, %rd`	10	rd	101000	0x14	0	0xA	-	rs2
`allocatex imm9, %rd`	`rdasr %asr20, %rd`	10	rd	101000	0x14	1	0xA	imm9	
`gets %rs2, N, %rd`	`rdasr %asr20, %rd`	10	rd	101000	0x14	0	0xB	N	rs2
`getg %rs2, N, %rd`	`rdasr %asr20, %rd`	10	rd	101000	0x14	0	0xC	N	rs2
`fgets %rs2, N, %fd`	`rdasr %asr20, %fd`	10	fd	101000	0x14	0	0xD	N	rs2
`fgetg %rs2, N, %fd`	`rdasr %asr20, %fd`	10	fd	101000	0x14	0	0xE	N	rs2
`ldbp %rd`	`rdasr %asr19, %rd`	10	rd	101000	0x13	0	0x1	-	00000
`ldfp %rd`	`rdasr %asr19, %rd`	10	rd	101000	0x13	0	0x2	-	00000
`crebas %rs2, %rs1`	`rdasr %asr19, %rs1`	10	rs1	101000	0x13	0	0x3	-	rs2
`crebas imm9, %rs1`	`rdasr %asr19, %rs1`	10	rs1	101000	0x13	1	0x3	imm9	
`crebis %rs2, %rs1`	`rdasr %asr19, %rs1`	10	rs1	101000	0x13	0	0x4	-	rs2
`crebis imm9, %rs1`	`rdasr %asr19, %rs1`	10	rs1	101000	0x13	1	0x4	imm9	
(reserved UTLEON3)	`wrasr %rs1, %asr20`	10	0x14	110000	rs1	0	0x1	-	00000
`setstart %rs1, %rs2`	`wrasr %rs1, %asr20`	10	0x14	110000	rs1	0	0x2	-	rs2
`setstart %rs1, imm9`	`wrasr %rs1, %asr20`	10	0x14	110000	rs1	1	0x2	imm9	
`setlimit %rs1, %rs2`	`wrasr %rs1, %asr20`	10	0x14	110000	rs1	0	0x3	-	rs2
`setlimit %rs1, imm9`	`wrasr %rs1, %asr20`	10	0x14	110000	rs1	1	0x3	imm9	
`setstep %rs1, %rs2`	`wrasr %rs1, %asr20`	10	0x14	110000	rs1	0	0x4	-	rs2
`setstep %rs1, imm9`	`wrasr %rs1, %asr20`	10	0x14	110000	rs1	1	0x4	imm9	
`setblock %rs1, %rs2`	`wrasr %rs1, %asr20`	10	0x14	110000	rs1	0	0x5	-	rs2
`setblock %rs1, imm9`	`wrasr %rs1, %asr20`	10	0x14	110000	rs1	1	0x5	imm9	
(reserved UTLEON3)	`wrasr %rs1, %asr20`	10	0x14	110000	rs1	0	0x6	-	rs2
(reserved UTLEON3)	`wrasr %rs1, %asr20`	10	0x14	110000	rs1	1	0x6	imm9	
`release %rs1`	`wrasr %rs1, %asr20`	10	0x14	110000	rs1	0	0x9	-	00000
`break`	`wrasr %g0, %asr20`	10	0x14	110000	00000	0	0xA	-	00000
`puts %rs2, %rs1, N`	`wrasr %rs1, %asr20`	10	0x14	110000	rs1	0	0xB	N	rs2
`putg %rs2, %rs1, N`	`wrasr %rs1, %asr20`	10	0x14	110000	rs1	0	0xC	N	rs2
`fputg %fs2, %rs1, N`	`wrasr %rs1, %asr20`	10	0x14	110000	rs1	0	0xD	N	fs2
`fputs %fs2, %rs1, N`	`wrasr %rs1, %asr20`	10	0x14	110000	rs1	0	0xE	N	fs2
`creba %rs1, %rs2`	`wrasr %rs1, %asr19`	10	0x13	110000	rs1	0	0x3	-	rs2
`creba %rs1, imm9`	`wrasr %rs1, %asr19`	10	0x13	110000	rs1	0	0x3	imm9	
`crebi %rs1, %rs2`	`wrasr %rs1, %asr19`	10	0x13	110000	rs1	0	0x4	-	rs2
`crebi %rs1, imm9`	`wrasr %rs1, %asr19`	10	0x13	110000	rs1	0	0x4	imm9	
`print %rs1, %rs2`	`wrasr %rs1, %asr19`	10	0x13	110000	rs1	0	0xF	-	rs2
`print %rs1, imm9`	`wrasr %rs1, %asr19`	10	0x13	110000	rs1	1	0xF	imm9	

Table D.6: General MT extensions for a SPARC v8 instruction set.

Non-prefixed values are in base 2, and values prefixed with "0x" in base 16.

Assembly pattern	bits 26-31	Format	bits 21-25	bits 16-20	bits 13-15	bit 12	bits 5-11	bits 0-4
`ldbp Rc`	0x01	Op	11111	11111	0	0	0000000	Rc
`ldfp Rc`	0x01	Op	11111	11111	0	0	0000001	Rc
`gettid Rc`	0x01	Op	11111	11111	0	0	0000010	Rc
`getfid Rc`	0x01	Op	11111	11111	0	0	0000011	Rc
`getpid Rc`	0x01	Op	11111	11111	0	0	0000100	Rc
`getcid Rc`	0x01	Op	11111	11111	0	0	0000101	Rc
`getasr imm8, Rc`	0x01	Op	11111	imm8		1	0000110	Rc
`getapr imm8, Rc`	0x01	Op	11111	imm8		1	0000111	Rc
`break`	0x01	Op	11111	11111	0	0	0001000	11111
(future local)	0x01	Op	x	x	x	x	000xxxx	x
`setstart Rf, Rv`	0x01	Op	Rf	Rv	0	0	0100000	11111
`setstart Rf, imm8`	0x01	Op	Rf	imm8		1	0100000	11111
`setlimit Rf, Rv`	0x01	Op	Rf	Rv	0	0	0100001	11111
`setlimit Rf, imm8`	0x01	Op	Rf	imm8		1	0100001	11111
`setstep Rf, Rv`	0x01	Op	Rf	Rv	0	0	0100010	11111
`setstep Rf, imm8`	0x01	Op	Rf	imm8		1	0100010	11111
`setblock Rf, Rv`	0x01	Op	Rf	Rv	0	0	0100011	11111
`setblock Rf, imm8`	0x01	Op	Rf	imm8		1	0100011	11111
`putg Rv, Rf, imm5`	0x01	Op	Rf	Rv	0	0	0100100	imm5
`putg imm8, Rf, imm5`	0x01	Op	Rf	imm8		1	0100100	imm5
`fputg Fv, Rf, imm5`	0x05	FP Op	Rf	Fv	0		0100100	imm5
`puts Rv, Rf, imm5`	0x01	Op	Rf	Rv	0	0	0100101	imm5
`puts imm8, Rf, imm5`	0x01	Op	Rf	imm8		1	0100101	imm5
`fputs Fv, Rf, imm5`	0x05	FP Op	Rf	Fv	0		0100101	imm5
`release Rf`	0x01	Op	Rf	11111	0	0	0101000	11111
(future remote async)	0x01	Op	x	x	x	x	010xxxx	x
(future remote async)	0x05	FP Op	x	x	x	x	010xxxx	x
`sync Rf, Rc`	0x01	Op	Rf	11111	0	0	0110000	Rc
`getg Rf, imm5, Rc`	0x01	Op	Rf	imm5	0	0	0110010	Rc
`fgetg Rf, imm5, Fc`	0x05	FP Op	Rf	11111	0		0110010	Fc
`gets Rf, imm5, Rc`	0x01	Op	Rf	imm5	0	0	0110011	Rc
`fgets Rf, imm5, Fc`	0x05	FP Op	Rf	11111	0		0110011	Fc
(future remote sync)	0x01	Op	x	x	x	x	011xxxx	Rc
(future remote sync)	0x05	FP Op	x	x	x	x	011xxxx	Fc
`allocate Rp, Ro, Rf`	0x01	Op	Rp	Ro	0	0	1000000	Rf
`allocate Rp, imm8, Rf`	0x01	Op	Rp	imm8		1	1000000	Rf
`allocate/s Rp, Ro, Rf`	0x01	Op	Rp	Ro	0	0	1000001	Rf
`allocate/s Rp, imm8, Rf`	0x01	Op	Rp	imm8		1	1000001	Rf
`allocate/x Rp, Ro, Rf`	0x01	Op	Rp	Ro	0	0	1000011	Rf
`allocate/x Rp, imm8, Rf`	0x01	Op	Rp	imm8		1	1000011	Rf
`creba/a Rs, Rv`	0x01	Op	Rs	Rv	0	0	1100001	11111
`creba/a Rs, imm8`	0x01	Op	Rs	imm8		1	1100001	11111
`creba/s Rs, Rv, Rf`	0x01	Op	Rs	Rv	0	0	1100001	Rf
`creba/s Rs, imm8, Rf`	0x01	Op	Rs	imm8		1	1100001	Rf
`crebi/a Rs, Rv`	0x01	Op	Rs	Rv	0	0	1110001	11111
`crebi/a Rs, imm8`	0x01	Op	Rs	imm8		1	1110001	11111
`crebi/s Rs, Rv, Rf`	0x01	Op	Rs	Rv	0	0	1110001	Rf
`crebi/s Rs, imm8, Rf`	0x01	Op	Rs	imm8		1	1110001	Rf
(future allocate)	0x01	Op	x	x	x	x	1xxxxxx	x
`crei Rf, imm16(Rp)`	0x03	Memory	Rf	Rp	imm16			
`cred Rf, imm21`	0x04	Branch	Rf	imm21				
`print Rs, Rv`	0x01	Op	Rs	Rv	0	0	0001111	11111
`print Rs, imm8`	0x01	Op	Rs	imm8		1	0001111	11111
`printf Rs, Fv`	0x05	FP Op	Rs	Fv	0		0001111	11111

Table D.7: MT extensions for the DEC/Alpha AXP 24264 instruction set.

Non-prefixed values are in base 2, and values prefixed with "0x" in base 16.

Appendix E

On-chip placement and distribution

Abstract

This appendix complements chapter 11 and details the thread mapping of logical threads to processors as implemented in the MGSim system emulator at the end of 2011.

Contents

E.1 General protocol

The requirement for explicit placement is closely related to resource management on chip. Indeed, bulk creation and thread contexts constitute physical resources over one or multiple processors; they are allocated and released via dedicated on-chip messages, which can be sent via processor instructions. Each bulk creation context can be used to activate families composed of concurrent logical threads; logical threads run on a set of parallel thread contexts allocated from physical resources on the same processors as the bulk creation contexts (section 4.2). Both family and thread contexts are thus finite resources local to each processor; a protocol to manually place computation to specific processors is therefore also a protocol that controls resource usage on chip.

We describe the proposed protocol here.

To start with, all execution units (processors, or "cores") are provided an address on an on-chip network. The "allocate," "create" and other concurrency operations from chapter 4 should be considered as *event messages* on this network.

A single family allocation event reserves a "virtual" family context spread over multiple cores, which can subsequently be bulk-created and bulk-synchronized. For this the program optionally provides a *placement size* along with the target processor address upon family allocation (section 4.3.2.1). If this size is larger than 1, the processor receiving the allocation may optionally negotiate a multi-core allocation with its neighbors (according to the topology of the network), trying to maximize the number of processors up to the size specified. As with the window size (section 4.3.2.2), parallelism is not guaranteed: the placement size is only an upper bound and the work can be performed even if only one processor is available, or if multi-processor allocation is not supported on that processor, regardless of the size specified.

When a multi-processor allocation completes, a bulk creation context has been reserved on each selected processor and all the contexts are linked to a master context selected among them. This context becomes the result of the original allocation event. When the subsequent creation event is sent to the master context, that processor forwards the creation to all participating processors; conversely, bulk synchronization on the master context effects bulk synchronization on all contexts. The master context is thus a *proxy* and provides the same interface as a single-processor family context to the issuing thread.

To support *implicit placement*, each bulk creation context further holds a default target processor and placement size, which is called its *default placement*. When a bulk creation context is allocated, the default placement is automatically set to the placement specified during allocation. When an allocation is issued by a program without an explicit placement, the default placement is used. This allows a program to automatically restrict the parallelism of a tree of implicitly placed concurrent families to a local cluster around the designated processor, bounded by the placement size.

Finally, the program can also parameterize the *logical thread distribution* of logical threads over processors. At least two distribution strategies are defined. The default is a *flexible distribution* which leaves the distribution unspecified, with the guarantee of a best effort to maximize the amount of parallelism by the specific architecture implementation (e.g. via work stealing). This allows a program to offload load balancing to the underlying hardware implementation. However, when the group of processors is known to be homogeneous, or when an explicit assignment is desired, the program can specify an *even distribution* explicitly which statically partitions the logical threads indexes evenly over the selected processors.

Address used	Automatically transformed to	Message sent to
0	*Default placement*	Implicitly, same processor cluster as where the issuing thread was placed
1	Processor address of the local processor, size 1, same capability	Implicitly, same processor as where the issuing thread is running
$C + 2P + S$	Processor address P, size S, capability C	Explicit target cluster

Table E.1: Address transformation for allocation messages on chip.

E.1.1 Global family identification

As families can be created across the chip and synchronized on from processors other than where they were created, the protocol defines Global Family Identifiers (GFIDs). These are composed of two parts: the address of the processor where the context is allocated (or the address of the master processor for distributed families) and the address of the context in the local structures of that processor.

To allow flexible control of bulk creation contexts and families by programs across arbitrary points on the chip, the address of a context can be observed and manipulated as a value. Its encoding is constrained to fit in the smallest common machine word size supported throughout the chip, in order to facilitate the integration in a software stack.

E.1.2 Isolation and capabilities

With regards to isolation, the protocol is extensible using capabilities [Lin76]. When these are used, each processor on the system holds a table of *processor access capabilities*, maintained by system-level resource managers, which filters incoming allocation messages. Programs must then request capabilities to said resource managers before they can issue "allocate." The capabilities need not be visible to programs, as it can be embedded either in a private memory at the requesting core (when dedicating entire cores to an isolated process) or the requesting family or thread context (when fine-grained access control is desired). Upon receiving an allocation message, a core allocation unit then generates a *family access capability* along with the GFID returned to the requesting thread. This must then be joined with the core capability upon subsequent creation, remote register accesses, synchronization and release messages. When a bulk creation context is allocated in one protection domain and used in another, the resource manager must be informed accordingly to propagate the access capabilities.

E.2 Reference implementation

The protocol described above has been prototyped in the system emulator for testing purposes, as follows:

- all the processors are numbered along a space-filling curve to ensure multi-scale space locality on the distributed cache network (fig. 3.9);

Side note E.1: Processor address decoding in the reference implementation.

The addressing scheme requires that all cluster sizes are powers of two, and that a cluster of any size S must start with a processor address multiple of S. The capability bits are placed on the MSBs unused by the processor address.
For example, the value A = 112040 on a chip with N = 64 processors, encoded in binary as `11011010110101000`, corresponds to:

- S = 8: find the first bit set from A's LSB, here `1000`,
- P = 16: compute $\frac{A-S}{2} \mod N$, here `010000`,
- C = 112000: compute $A \mod 2N$, here `11011010110000000`.

Side note E.2: About the distribution of families with "shared" channels.

For families of threads using "shared" synchronizers (section 4.3.3.3), all threads are always created at the first processor of the target cluster, although their default placement may be configured to the entire cluster. The reason why local threads for this kind of family are not spread over multiple cores stems from an observation when such an implementation was tried. If such families were distributed over multiple processors, the first logical thread which reads from an input "shared" channel on every processor would wait for the corresponding output by from the last logical thread on the previous processor. This would effectively force a sequentialization of the entire communication chain and prevent multi-processor scalability altogether. As this effect was likely to cause programs to avoid using the feature, the extra effort required to implement the feature was not invested.

```
1 static inline size_t sl_placement_size(sl_place_t C)
2 {
3   // erase all bits set but the rightmost, i.e. the size
4   return C & -C;
5 }
```

Listing E.1: Placement computation that extracts the size from a virtual cluster address.

- the placement operand provided to the "allocate" operation (section 4.3.2.1) is handled by the hardware as per table E.1 and side note E.1.
- the implementation supports the even distribution strategy of logical threads to processors in distributed families, but only for families without "shared" channels (cf. side note E.2);
- capabilities are decoded but not verified (yet).

Also, new assembler mnemonics "`getpid`" and "`getcid`" are introduced which map to machine instructions that read the default placement and the local processor address of the issuing thread, respectively.

E.3 SL primitives

To control this protocol, the primitives in listings E.1 to E.6 and table E.2 are available in a SL library header file.

Primitive	Alpha instructions	SPARC instructions
`sl_placement_size`	2	2
`sl_first_processor_address`	2	2
`split_upper`	5	5
`split_lower`	5	5
`at_core`	7	6
`split_sibling`	11	11
`next_core`	13	13
`prev_core`	13	13

Table E.2: Execution cost of the placement primitives.

All instructions can be performed locally in the ALU in one pipeline cycle.

```
1 static inline sl_place_t sl_first_processor_address(sl_place_t C)
2 {
3   // erase the rightmost bit set, i.e. the size
4   return C & (C - 1);
5 }
```

Listing E.2: Placement computation that extracts the absolute address of the first core in a virtual cluster.

```
1  static inline sl_place_t split_upper(void) {
2      sl_place_t d = sl_default_placement();
3      size_t size = sl_placement_size(d);
4      return d + size / 2;
5  }
6  static inline sl_place_t split_lower(void) {
7      sl_place_t d = sl_default_placement();
8      size_t size = sl_placement_size(d);
9      return d - size / 2;
10 }
```

Listing E.3: Placement computation that divides the current cluster in two and addresses either the upper or lower half.

```
1 static inline sl_place_t at_core(unsigned int P) {
2    sl_place_t d = sl_default_placement();
3    sl_place_t f = sl_first_processor_address(d);
4    return (f + P * 2) | 1;
5 }
```

Listing E.4: Placement computation to place all the created thread at a core offset P within the local cluster.

```
1  static inline sl_place_t split_sibling(void) {
2      sl_place_t d = sl_default_placement();
3      size_t size = sl_placement_size(d);
4
5      sl_place_t lp = sl_local_processor_address();
6      sl_place_t mybit = (lp * 2) & size;
7      sl_place_t otherbit = mybit ^ size;
8
9      sl_place_t f = sl_first_processor_address(d);
10     return f | otherbit | (size / 2);
11 }
```

Listing E.5: Placement computation that divides the current cluster in two and addresses the other half relative to the current core.

```
1  static inline sl_place_t next_core(void) {
2      sl_place_t d = sl_default_placement();
3      size_t size = sl_placement_size(d);
4
5      sl_place_t lp = sl_local_processor_address();
6      // make relative to start of cluster
7      sl_place_t rel_lp = lp & (size - 1);
8
9      // next core address:
10     sl_place_t rel_np = (rel_lp + 1) & (size - 1);
11
12     sl_place_t f = sl_first_processor_address(d);
13     return (f + rel_np * 2) | 1;
14 }
15 static inline sl_place_t prev_core(void) {
16     // same as ``nextcore'' with
17     // (rel_lp - 1) instead of (rel_lp + 1).
18 }
```

Listing E.6: Placement computation to place all the created thread at the next or previous core within the local cluster.

Appendix F

Semantics of objects in the C language

Abstract

The process of extending the C language for use with a new processor architecture with hardware support for synchronization channels requires us to carefully avoid clashes between the hardware semantics and the (well-specified) semantics of the C abstract machine. As a foundation to this work, we analyzed the specification in [II99, II11b] to uncover and summarize how the C language handles data in programs. To our knowledge, no such analysis exists in previous work. Our findings are detailed here.

Contents

Side note F.1: About the concept of objects in the C language specification.

While objects are a central concepts in the abstract semantics of C, they are only indirectly defined throughout [II99, II11b] and similar documents. An object is any of the following: an entity defined by a declaration, an entity defined by a non-array function parameter declaration, a constant (character/integer, floating-point or literal string), the value of an initializer, the value of an enumeration tag, an allocated object, the result of a computation in an expression, the return value of a function call. The following are not objects despite the possibility of taking their address: functions, labels (in some implementations).

Side note F.2: About immutable objects.

The following objects are immutable: constants, the value of initializers, the value of enumeration tags, temporary results in expressions, return values of function calls, objects defined with the `const` qualifier, non-array function parameters declared with the `const` qualifier; the following objects are mutable: objects defined without the `const` qualifier, non-array function parameters declared without the `const` qualifier.

Side note F.3: About objects without an address.

The following objects cannot have their address taken: objects defined with the `register` storage-class qualifier, temporary results in expressions, return values of function calls, constants, the value of initializers, the value of enumeration tags; the following objects can have their address taken: objects defined without the `register` storage-class qualifier, non-array function parameters, allocated objects.

F.1 Fundamental characteristics

An *object* is an entity able to store data for a C program.

A C object *exists* thoughout its *lifetime*.

The following *properties* of objects are *defined and fixed* throughout their lifetime:

- whether the object is *mutable* or not;
- whether the object *can have its address taken* or not;
- its *size*.

The properties of an array considered as an object always extend as properties of its individual items considered as separate objects.

The layout in storage of an object is identical to an array of N objects of type char (N being the size of the object as per `sizeof`). This array is the representation of the object.

If an object is mutable, then each char of its representation can be muted (modified) individually multiple times, and a read to each will consistently and deterministically yield the value that was last stored before the immediately preceding sequence point. The initial value of each char of the representation, although unspecified, exists and can be read.

If the address of an object can be taken, then:

- its address is constant throughout its lifetime,
- it is possible to compute the address of each individual char of its representation,
- the addresses of each individual char are contiguous,

Side note F.4: About object sizes.

The size of an object is derived from the type used to define the object, or an expression for allocated objects. Note that the size is fixed even for allocated objects: the `realloc` library call semantically returns a new (distinct) object, although its address may be the same as the object given as input.

Side note F.5: About array item properties.

In particular: the items of a mutable array are separately mutable; the items of an immutable array are separately immutable; the items of an array that can have its address taken can have their address taken separately; the items of an array that cannot have its address taken cannot have their address taken separately; the size of an array is the size of an individual item multiplied by the number of items in the array.

Side note F.6: About multiple accesses between sequence points.

If an object is written to, then read between two adjacent sequence points, e.g. in `p[i++] = ++i`, the behavior of the program is undefined.

Side note F.7: About the initial char representation of objects.

The chars of the object representation can be read. In particular, the behavior of the following program piece is well-defined and consistently deterministic: `int x; x = x ^ x;`.

- there exist a valid address one past the last char in its representation.

No two distinct mutable objects that can have their address taken have overlapping representations.

It is possible to constructs objects of arbitrarily large sizes (as long as matching pointer types, `size_t` and `ptrdiff_t` are provided by the implementation).

F.2 Basic operations

Two basic operations are defined on objects and are valid throughout their lifetime:

- *read* for all objects;
- *write* for mutable objects.

The "read" operation has the following properties:

- it always completes within a finite amount of time before the next sequence point is reached;
- it yields the *value* stored in the object, according to the type of the designator used to access the object.

The "write" operation has the following properties:

- it always completes within a finite amount of time before the next sequence point is reached;
- it modifies the object so that further reads will yield the new value being stored.

Side note F.8: About valid addresses one past the last char.

C provides this guarantee in order to be able to consistently compute sizes by performing pointer arithmetic. A side-effect (arguably harmless) of this guarantee is that no object can occupy the last char in an address space.

It is worthy to note that reading from and writing to an existing object has no influence on the control flow: considering a read and/or write operation, then in all runs of the program where the sequence point immediately before is reached, if the object exists at that point the sequence point immediately after is reached in sequence within a finite amount of time. Of course, if the object does not exist at that point, or if a write operation is attempted on an immutable object, the behavior is undefined.

F.3 Syntax in the C language

The "read" operation that operates on objects is exposed implicitly in the language when objects are designated in expressions.

The "write" operation that operates on objects is exposed by the various assign operators and the increment/decrement operators.

F.4 Space for optimization

The C specification indicates that an expression needs not be evaluated if its value is not used and it does not have side-effects ([II99, 5.1.2.3§3], cite[5.1.2.3§4]isoc11).

This can be reworded as follows:

- if two writes to an object are expressed in a program and it can be proven during compilation that no other read to the same object ever takes place (at run-time) between the two writes, then the program can be transformed to an equivalent program where the first write is omitted; for example:

```
1 x = a; // can be omitted if x is
2        // not read before "x=b"
3        // below.
4 /* ... */
5 x = b;
```

- if two reads to an object are expressed in a program and it can be proven during compilation that no write to the same object ever takes place (at run-time) between the two reads, then the program can be transformed to an equivalent program where the second read is omitted and the value of the first read is reused instead; for example:

```
1 a = x;
2 /* ... */
3 b = x; // can be expressed as
4        // "b=a" if a and x were
5        // not written to.
```

- if a write to an object and a read to the same object are expressed in a program and it can be proven during compilation that no write to the same object ever takes place (at run-time) between the write and the read, then the program can be transformed to an equivalent program where the read is omitted and the value used in the first write is reused; for example:

```
1 a = x;
2 /* ... */
```

Side note F.9: Objects without primary designators.

In some implementations it is possible to define an object with a scoped lifetime, a known type but without a primary designator, e.g. anonymous (unused) function parameter declarations.

Side note F.10: Primary designator aliases for immutable objects.

The C specification leaves space for an implementation to share an immutable object across multiple primary designators, e.g. for string literals.

```
3 b = a; // can be expressed as
4        // "b=x" if a and x
5        // were not written to.
```

This optimization path is used routinely in most production-grade C compilers.

F.5 Lexical binding of lifetimes and resource allocation

The lifetime of an object is either *scoped* or *unscoped*. An object has unscoped lifetime if it is allocated. *All non-allocated objects have scoped lifetimes.*

Scoped lifetimes are bound to a *declaration* (either an object definition or a non-array function parameter declaration) and a *syntactic scope*, and enforced by the environment according to completely specified semantics. Unscoped lifetimes are managed explicitly by the (running) program.

Whether an object with a scoped lifetime exists or not at any given point in a program is *entirely decidable during compilation.* Objects with scoped lifetimes have *all their properties known and fixed by their declaration.* These two properties allow a compiler to *statically allocate resources* for all objects with scoped lifetimes.

F.6 Primary designators

Objects with scoped lifetimes have a type and zero or one *primary designator* (derived from their definition).

No two distinct primary designators that designate mutable objects designate the same object.

As scoping also defines the visibility of primary designators, the portion of a program where a primary designator is visible is always a subset of the lifetime of the object it designates.

The main purpose of the design of primary designators in C is to *uniquely identify* objects.

Objects with unscoped lifetimes do not have a primary designator, always are mutable, always can have their address taken and have a size equal to the value of the argument of the function call used to start their lifetime.

F.7 Secondary designators

Any object can have zero or more *secondary designators* each with their own types (aliases).

For example, in the following program fragment:

Side note F.11: About array designators in function parameter lists.

This the main distinction between array and non-array function parameter declarations: non-array function parameters are primary designators to distinct objects from the argument in the caller, whereas array function parameters are secondary designators to the same object as the argument in the caller.

```
1 int a;
2 int *p = &a;
3 (a);
4 (*p)
```

There exists one object with primary designator `a` and secondary designator `*p` (and another object with primary designator `p`).

Secondary designators are *constructed* by the program *at run-time*, using only the following operations:

- from an existing designator, using the unary `&` operator;
- from an existing designator, using a cast expression;
- from an existing array designator, using the designator in a context where it is converted automatically to a pointer (either during pointer arithmetic, assigment to pointer, or when passing an array as a function argument);
- from an existing array designator, using the array subscript notation;
- from an existing union or structure designator, using the field subscript notation.

Secondary designators are *used* by the program using only the following syntax:

- for cast notations, using the cast notation directly;
- for field subscript notations, using the field subscript notation directly;
- for array subscript notations, using the array subscript notation directly;
- for secondary designators derived as pointer values (either by the unary `&` or using an array designator in a pointer context), using the unary `*` operator or the array subscript notation on a designator with pointer type to an object containing that pointer value.

The visibility of secondary designators can extend past and before the lifetime of an object. The behavior of a program that references an object via a secondary designator outside of its lifetime is undefined.

Secondary designators can designate any contiguous sequence of chars in the representation of an object; if the object is mutable any modification through a secondary designator only modifies the bytes designated by the designator as constrainted by its type.

While it is possible to construct a secondary designator to an object defined with `const` so that the secondary designator does not have the const qualifier, any attempt to modify an object through such a designator still has undefined behavior ([II99, 6.7.3§5], [II11b, 6.7.4§6]).

F.8 The fine print in the C semantics

- Any expression expressed in C is a designator for an object; however while some expressions are secondary designators for an object that already exists, some other

expressions are primary designators for object defined through them (e.g. intermediary computation results). Which expressions are primary designators and secondary designators is entirely specified.

- The C syntax does not allow a program to designate an array item object without taking the address of the array. Due this feature, an array object which cannot have its address taken (like when `register` is used) cannot be used in any useful way.
- If an object cannot have its address taken, then:
  - the only kind of secondary designator that can be constructed for the object are (compatible) cast expressions and union and structure field accesses; in particular, array subscript and automatic conversion of arrays to pointers are not possible (by definition) for arrays that cannot have their address taken;
  - it is always possible to identify the object, its primary designator (if any), its lifetime and its other properties (mutability, size) from any of its secondary designators during compilation;
  - the visibility of any secondary designator to the same object is always a subset of the object's lifetime;
  - it is possible to identify all its secondary designators during compilation.

  These properties extend to any object, in a given program, that *is proven, within that program, to not have its address taken* (even if it could be taken). This says that if the property "a specific object does not have its address taken" is verified to be true for a given object in a program up to a specific point, then the properties above hold for that object up to the same specific point.
- Non-array function parameter declarations are primary designators to objects whose initial value is a copy of the value of the object designated by the argument in the caller, whereas array function parameter declarations are secondary designators to the object designated by the argument in the caller.
- While the qualifiers on a secondary designator constrain what operations can be expressed in the language using that designator (for example, a designator with the `const` qualifier cannot be used for a write operation), they do not have any impact on the essential mutability of an object. If the object is mutable, then at least one of the following always hold:
  - the non-const qualified primary designator is visible (then it can be used to write to the object),
  - a new non-const qualified secondary designator can be constructed from a const-qualified secondary designator and used for write operations with the usual operational semantics of writes.

  In particular, the program in listing F.1 is valid.

```c
void foo(const int a[]) {

 // here ``a'' is a const-qualified secondary
 // designator for the array object given as
 // argument by the caller.

 // so we can construct a new secondary
 // designator to drop the const qualifier
 // and change the type.
 char *p = (char*) &a;

 // the following is always valid if
 // the array in the caller has at least
 // one item and is mutable.
 a[0] = 10;
}

int main(void) {
 int a[10];
 foo(a); // valid call, first byte of
         // array is modified
 return 0;
}
```

Listing F.1: Const designator to a mutable object.

Appendix G

Original language interface
—Description and defects

Abstract

Prior to our work some research had been realized to design a C-based interface to the proposed architecture. This appendix reviews this previous work and provides a comprehensive description of the corresponding language design. Through analysis, we uncover hidden complexity in this prior language design, and provide a proof that the language cannot be compiled to some variants of the target architecture interface from chapter 4.

Contents

Program	Description
`fibonacci`	Prints the first values of the Fibonacci sequence (the number of values is statically configured).
`frame-invert`	Computes the "negative" of a 2D, 8bpp black picture represented as an Iliffe vector [Ili61] (a one-dimensional array of pointers to one-dimensional arrays of pixel for each row in the image) of static size, and prints the result.
`matmul`	Computes the matrix-matrix product of static square integer matrices (static size and values), and prints the result.
`sine`	Computes the sine of a floating-point value using a 9-level Taylor expansion, and prints the result.

Table G.1: Example programs using the proposed C extensions, December 2008

Key	Title
[Jes06a]	μTC - an intermediate language for programming chip multiprocessors.
[Jes06b]	Microthreading, a model for distributed instruction-level concurrency.
[BJK07]	Strategies for compiling μTC to novel chip multiprocessors.

Table G.2: Academic publications related to the C language extensions, December 2008

G.1 Analysis of pre-existing sources

The process to re-construct the prior art was bootstrapped by a preliminary analysis of what information was already available. The following is an exhaustive list of the sources we found relevant to the language extensions at that point (late 2008):

‹i› a library of 4 short example programs written using the proposed primitives, listed in table G.1;
‹ii› 4 academic publications containing descriptions of the language constructs and examples, listed in table G.2;
‹iii› an internal (unpublished) technical report describing the language extensions [LB08];
‹iv› program source code for "`utc2cpp`," a translator from the proposed language to a software-based concurrency run-time system [vTJLP09] with different semantics from the proposed hardware architecture;
‹v› the various authors of the previous sources themselves.

During this research, we also found 7 additional example programs using the proposed language extensions, developed in a separate project (ÆTHER). However, since these programs use additional concurrency semantics that are not supported by the proposed architecture (chapters 3 and 4), they are not further considered here.

G.1.1 Previously consistent statements

Close study of the aforementioned sources reveal the following consensus throughout:

- a new language construct based on the word "`create`" is added to the C language, with a syntactic structure similar to C's `for` construct, and which can appear in C code in the same syntactic context as other *statements*;

```
1  #define N 100
2  int A[N], B[N];
3
4  thread scale1(void) {
5      index int i;
6      B[i] = A[i] * 2;
7  }
8
9  ...
10 {
11     int f1;
12     create(f1; 0; N; 1; 0) scale1();
13     sync(f1);
14 }
```

Listing G.1: Example code using the "create" construct and separate thread function to scale a vector.

- the "`create`" constructs describes a concurrent computation composed of multiple instances, called *logical threads*, of a *thread program* described by a C *syntactic block* (text between matched curly braces `{` and `}`);
- the overall structure of the new construct is the word "`create`," followed by *concurrency parameters* between parentheses, followed by a description of the work to do;
- the thread program can be specified as a syntactic construct analog to a C *function definition*, but including the special word "`thread`" in the function declarator;
- the concurrency parameters include the *start*, *limit* and *step* values that determine the number of logical threads and the logical thread index range (cf. section 4.3.2.3), as well as a *block* value that constrains the maximum breadth of actual parallelism (cf. section 4.3.2.2);
- in the program description of the basic computation unit, the word "`index`" can be added to an integer variable declaration to signify that this variable will be automatically initialized by the processor to the logical thread index;
- in the surrounding syntactic block, the control flow is synchronized with the completion of the concurrent computation at the point where the word "`sync`" appears;
- the occurrences of the words "`create`" and "`sync`" are matched together in the program code by the use of a common lexical identifier in the surrounding program text, which is to be assigned a "*family identifier*" value upon the creation of the concurrent computation;
- the unit of work described by the composition of "`create`," "`sync`" and the related syntax can be replaced by a `for` loop in C without changing the computation semantics of the program;
- the special value 0, when specified for the *block* parameter, indicates that the underlying architecture is free to select the effective amount of parallelism when instantiating the work;
- each thread program can itself contain uses of "`create`" and "`sync`," to allow the hierarchical composition of multiple levels of concurrent computations at run-time.

An example code that illustrates these statements is given in listing G.1. In this example, two static arrays A and B are defined in the global scope. Then a thread program

```
1 thread innerprod(int* X, int* Y, shared int s) {
2     index int i;
3     s = X[i] * Y[i] + s;
4 }
```

Listing G.2: Example thread program using a channel interface specification.

that describes one instance of a concurrent computation is defined using the word "`thread`," using also "`index`" to declare the discriminating logical index `i`. This thread program can then be *used* in another computation block with "`create`" followed by the concurrency parameters between parentheses, followed by the name of the thread program. Then the word "`sync`" indicates that the control flow of the surrounding block should be suspended until the concurrent parallel computation completes (bulk synchronization). The concurrency parameters are specified by *positional parameters*, that is, their order specifies their role: the *start* value appears first, followed by the *limit*, *step* and *block* parameters. The name of the variable assigned the family identifier is listed as first parameter for both "`create`" and "`sync`."

In this example, the basic thread program "scale1" only assumes that the index variable is automatically initialized and that a memory interface (load, store) to the arrays A and B is available. In particular, the base addresses of the arrays A and B are not a dynamic input to the basic computation: the position of the array definitions in the global scope imply that the linker program will rename statically all uses of the names "A" and "B" in the code to constant array locations in memory.

To define additional input and output channels to a basic thread program, the proposed language extension allows programmers to declare *thread parameters* in the definition of a thread program. All the sources isolated above agree on the following consensus:

- when used with the word "`thread`," the syntactic position of function parameter declarations in the function definition (after the function name, between parentheses) is used to define the *interface* of the thread program;
- each position of the interface specification declares either one or two separate communication endpoint for the computation;
- at each position of the interface specification, the *name* used in the declarator will subsequently refer to the corresponding communication endpoints;
- at each position of the interface specification, the word "`shared`" can be added to distinguish declarations of "global" channels (where all concurrent instances of the computation must share the same source) from "shared" channels (where the endpoints are daisy-chained from one instance to the next). This is intended to correspond to the semantics of the hardware machine interface (cf. section 4.3.3);
- in the thread program, occurrences of the channel endpoint names are intended to be translated to *input* operations (for "global" channels) and either *input or output* operations (for "shared" channels), depending on whether the name is used as read-only operand of another C construct, or as the target of an assignment.

Remarkably, any channel declaration with the word "`shared`" simultaneously declares two channel endpoints, one for inputs and another, distinct one for outputs.

An example is given in listing G.2. In this example, the definition of the thread program "innerprod" declares 4 channel endpoints. The first two, designated by the names "X" and

"Y," correspond to "global" channels in the underlying architecture, i.e. the assumption that all concurrent instances of the computation unit will share the same data source. The third position, where the word "`shared`" is used, simultaneously declares two endpoints with the common name "s." The input part is used when the word "s" is subsequently used as an input operand in an expression, whereas the output part is used when the word "s" is used as the target of an assignment. The only statement in the program block can be decomposed as follows:

1. read the first array address from channel "X";
2. compute the array element address "X[i]" using the logical thread index;
3. issue a load from memory using the address "X[i]";
4. read the second array address from channel "Y";
5. compute the array element address "Y[i]";
6. issue a load from memory using the address "Y[i]";
7. wait on completion of both memory loads, and add the results;
8. read from the input channel corresponding to the name "s";
9. add the value read to the intermediate sum;
10. write the result to the output channel corresponding to the name "s".

Finally, the design called for a means to relate the channel endpoints of the thread program to the data items in the surrounding code at the point of use of the "`create`" construct. This was done as follows:

- in the "`create`" construct, after the specification of which thread program to run, the program can specify a list of positional *thread arguments* with the a syntax similar to function calls: a comma-separated list of arguments enclosed in parentheses;
- each positional argument corresponds one-to-one to the endpoints of the channel interface specified in the thread program definition;
- for each position corresponding to a "global" channel, the program can specify any value, to be subsequently propagated by the substrate architecture as the source of input operations for all concurrent instances of the computation unit;
- for each position corresponding to a "shared" channel, the program can specify a *lexical identifier* referring to a C variable declaration in the surrounding scope. The value of this variable at the point the "`create`" construct is reached during execution is to be propagated by the architecture as the source of input operations by the first concurrent instance of the computation unit; and the value output by the last instance is to be propagated as the new value of the C variable after the control flow passes the "`sync`" word.

An example is given in listing G.3, which complements listing G.2. In this program fragment, the "`create`" construct uses the 3 positions of the interface defined by the thread program "innerprod." The first two positions, corresponding to the "global" channel declarations X and Y in the definition of "innerprod," are provided the computed values &A[0] and &B[0], which describe the start addresses in memory of the arrays A and B, respectively. The third position, corresponding to channel "s" in "innerprod", is provided the identifier "res" which names a variable declared in the same scope. The value of "res" at the point the "`create`" construct is reached is determined by the assignment that immediately precedes (0 in this case), and this is the value sent as input to the first instance of the computation. When the last instance completes, the value it writes to its output channel for "s" is then

```
1  #define N 100
2  int A[N], B[N];
3
4  ...
5  {
6     int res, f1;
7     ...
8     res = 0;
9     create(f1; 0; N; 1; 0) innerprod(&A[0], &B[0], res);
10    sync(f1);
11    printf("The sum is %d\n", res);
12 }
```

Listing G.3: Example use of the "innerprod" thread program.

propagated back to "res" in the invocation context. In the example this value is then printed to reveal the result of the computation.

G.1.2 Committed but unsupported features

The description given in the previous section is sufficient to explain the program code in source ‹i›; moreover, it is consistent with all the code fragments given as illustrations in source ‹ii›.

In this section, we provide the *additional* statements about the language design that have been stated in sources ‹ii› to ‹iv› which were not directly illustrated by example programs. Because they were stated in writing and agreed upon, they were assumed to be part of the language specification by the research community. However these are also the point where the language design became inconsistent with the hardware design.

The first additional statement concerns the relationship between the words "`create`" and "`sync`" in the language syntax. While the first publications [Jes06a, Jes06b] and source ‹i› are silent on this topic, the paper [BJK07] and sources ‹iii› and ‹iv› indicate that:

- the construct starting with the word "`create`" and ending with the thread argument specification and a semicolon can be used in the syntactic place of a C statement, and
- the construct starting with the word "`sync`," followed by a family identifier variable between parentheses and a semicolon, can also be used in the syntactic place of a statement, albeit at a *different statement position* in the program than the "`create`" construct.

In other words, this additional formulation *decouples* the "`create`" and "`sync`" part of a concurrent work definition. Although no example was given alongside this statement, its consequences are clear: the design allows the expression of other statements, control flow structures or blocks in the program text between the "`create`" and "`sync`" parts. The motivation of this clarification was twofold. The first was increase the amount of concurrency that could be expressed, by allowing the description of extra computations that could be carried out simultaneously with the concurrent work described by "`create`," before the "`sync`" construct is reached by the control flow. However, the main motivation for this extra step was to also state that:

Side note G.1: Attempt to constrain the well-formedness of programs.

Interestingly, the last part of the quoted statement, which tries to establish a criterion for the well-formedness of programs, actually fails to do so. Indeed, for any *actual* execution of a thread program containing both "`create`" and "`sync`" constructs, the execution can be described as a *single* sequence of intermediate steps which describe how the control flow *actually* passes all sequence points in the program[1]. In any such sequence, observed at the point in time "`sync`" occurs, at most one "`create`" antecedent will have assigned a value to the same family identifier variable prior to that point. Because the execution of a thread program is (conceptually) sequential, there is no circumstance where more than one "`create`" produces the family identifier value eventually consumed by "`sync`." Therefore, this last sentence, while logical, provides no sensical criterion to determine whether a program is well-formed or not.

- for any given "`create`" construct, the corresponding "`sync`" construct is optional; that is, a program can define an *asynchronous* computation which is not waited upon by the creating thread.

This extra step forward in the design was pressured by project partners during the separate project ÆTHER, as it was perceived to be required to implement a run-time system for the coordination language S-NET [GSS10, JS08], later revisited in [HÏ0, Chap. 7]). The step was made and the corresponding language feature implemented in the software-based run-time system [vTJLP09] that was used as prototype before the Apple-CORE project started. However, this decision was taken independently from the architecture research that led to the design we describe in chapters 3 and 4.

Finally, in source ‹iii› (Section "Microthread C," sub-section "sync"), we found the only further constraint on the relative placement of the constructs in the text of a program:

> "The sync function must not be used outside of a thread function and the argument must identify a variable [...] defined in the scope of the thread function in which the sync occurs and must under all circumstances be the direct result of a single create statement in the same scope as the sync. That is, the program is malformed if there exists the possibility that the sync is used on [...] the result of different creates."

The main message of this statement is that

- both "`create`" and "`sync`" constructs pertaining to the same variable holding a family identifier must appear in the same scope.

This statement further prevents a program to be structured in such a way that a concurrent work unit is created in one thread, and synchronized upon in a different thread, as this feature was not available in the underlying hardware architecture.

G.1.3 Overly restrictive statements

In source ‹iii› (Section "Microthread C," subsection "Program Structure"), we further find that

[1] This is opposed to the abstract view of the control flow graph as a whole, which considers simultaneously all possible paths through the graph.

```
1 void bar(int *p) {
2     *p = 10;
3 }
4
5 thread foo(shared int x) {
6   int *p = &x;
7   bar(p);
8 }
```

Listing G.4: Using the "address of" operator on a channel endpoint.

> "There are no regular C function calls allowed in [the proposed language extension]. All [functions] must be defined as thread functions and 'called' via the create action."

This conflicts with the requirements of generality set forth in section 6.2.1.

G.2 Hidden complexity in the proposal

To gain further understanding of the proposed language semantics, we can attempt to craft programs which exercise specifically what the documented prior work did *not* state about the new constructs. This revealed additional, *hidden complexity* in the design which was not envisioned by the research group previously. These are described here.

G.2.1 Nature of thread parameters

This code has no meaning with the target machine interface. Indeed, the communication channel endpoints are hardware synchronizers which cannot be indirectly addressed (section 4.2).

In more general terms, thread parameter names are not designators for C objects. As per the terminology introduced in Appendix F, the endpoints of communication channels do not have storage, so they do not fall under any of the categories for objects in C. Moreover, [II99, II11b] indicate that "an identifier can denote an object; a function; a tag or a member of a structure, union, or enumeration; a typedef name; a label name; a macro name; or a macro parameter." By introducing a new concept which does not fall in any of these categories, the "thread parameter," the proposal thus requires to extend the definition of *identifiers* in C and the semantics of identifiers in expressions. We must first extend the following properties of identifiers:

- about *scope*, [II99, 6.2.1§2] and [II11b, 6.2.1§2]: thread parameter identifiers have *block scope*, over the extent of the thread program body;
- about *name spaces*, [II99, 6.2.3] and [II11b, 6.2.3]: thread parameter identifiers are *ordinary identifiers*, i.e. they are in the same name space as ordinary variables, base types, typedef names and enumeration constants.

Then we consider that C only allows identifiers that refer to objects and functions in expressions ([II99, 6.5.1§2] and [II11b, 6.5.1§2]). This needs to be extended; however we must be careful in doing so: it is not sufficient to specify that "thread parameter identifiers

are primary expressions," we must also indicate *how they are converted to a value* that can be computed upon. To do this, we start by looking at the way C gives values to identifiers for objects and functions:

- 6.5.1§2 indicates that an identifier that designates an object is an lvalue; then 6.3.2.1§2 indicates an lvalue which does not have an array type is converted to the value stored in the designated object (and is no longer an lvalue); 6.3.2.1§3 indicates that an lvalue which designates an array is converted to a pointer to the start of the array (and is no longer an lvalue).
- 6.3.2.1§4 indicates that function designators, when used in expressions, are converted to a pointer to the function.

None of these statements apply to thread parameters; in particular thread parameters are not lvalues. Based on the sources we have isolated, we construct the following additional statement:

- identifiers for thread parameters are primary expressions; identifiers for "global" channels are converted to the value read from the corresponding channel endpoint; identifiers for "shared" channels are converted to the value read from the input endpoint of the corresponding channel pair; in both cases they are not lvalues.

With this statement, an expression of the form "x + 3" where "x" denotes a thread parameter becomes meaningful and has a value. Meanwhile, an expression of the form "&x" is disallowed, because "x" is converted to a non-lvalue first, so the & operator cannot be applied any more.

However, this is not sufficient to provide a meaning to all examples shown so far. Indeed, since thread parameter identifiers are not lvalues, they cannot appear at the left hand side of assignment operators ("=" "+=" etc., as per [II99, 6.5.16§2] and [II11b, 6.5.16§2]). A new meaning is thus required to allow expressions of the form "x = x + 3" where "x" denotes a thread parameter.

Most of the extra complexity lies here: the entire semantics of assignments in C require an object on the left-hand side of the operator. The constraints about typing, the semantics of the assignments regarding object representation and overlap, etc., are defined with the assumption that the target of the assignment is an object. To provide meaning to the examples, we must extend the definition of assignments to signify the output of a value on the outgoing endpoint of "shared" channels.

Moreover, the main definition of assignments ([II99, 6.5.16§3] and [II11b, 6.5.16§3]) indicates that the value of the entire assignment is the value of the left operand after the assignment. This is meaningless in our setting, and even quite dangerous, since the input endpoint of a "shared" channel may not be readable any more after the output endpoint has been written to (side note G.2).

In order to address this situation, the only way we found to give a meaning to the examples that would be otherwise compatible with existing C semantics is to create a *new statement* dedicated to the outgoing communication over "shared" channels:

- Syntax: an identifier that designates a thread parameter for a "shared" channel, followed by the assignment operator, followed by an expression, followed by a semicolon, is a *shared channel assignment*; this extends the syntax of expression statements ([II99, 6.8.3] and [II11b, 6.8.3]):

Side note G.2: About the input availability of "shareds" after writes.

With the "hanging" synchronizer mapping introduced in section 4.3.3.3, the endpoints of the outgoing "shared" channels to the first thread and the incoming "shared" channels from the last logical thread in a family are mapped to the same synchronizers in storage. This means that any value populated for the first created thread is overwritten by outputs performed by the last thread. If there is only one thread in the created family, this implies that the input endpoint in the created thread cannot be read reliably after the output endpoint is written to (the read will not produce the original value).

expression-statement:
identifier = *expression* ;
expression$_{opt}$;

This definition introduces a syntactic ambiguity between the new shared channel assignment, and assignment expressions where the left hand side lvalue is an object identifier. However the ambiguity can be resolved by disambiguating identifiers during lexical analysis, by ensuring that thread parameter identifiers have their own lexical class. No conceptual difficulty is added here, because lexical disambiguation of identifiers is already a prerequisite of C (e.g. to disambiguate "x * y;" which can be either an expression if the first identifier refers to an object, or a declaration if the first identifier refers to a type).

- Semantics: a shared channel assignments emits the value of the expression on its right hand side to the outgoing channel endpoint of the "shared" channel pair designated by its left hand side.

We can then simply extend this definition to compound assignment operators ("+=", etc) by establishing an equivalence between the form "x += *E* ;" and the form "x = x + *E* ;"

To summarize, we found that proper handling of thread parameters requires a new abstract concept in the C language, an extension of the definition of *identifiers*, and a new *statement* dedicated to communicate through the outgoing channel of a "shared" channel pair. This was not visible in the original language extension proposal.

G.2.2 Addressability of thread arguments

Consider the example fragment in listing G.5. This fragment appears valid. The "`create`" construct defines a concurrent unit of work using the thread program "foo." It also properly uses a local variable identifier as a thread argument for the "shared" channel, which provides the declared initial value of "x" (5) as the source value for the computation. Later in the program code the "`sync`" construct is used with the valid family identifier variable to wait on termination of the concurrent computation, and the final value of "x" is only assigned to "y" after the control flow passes "`sync`."

However, what is the value of variable "y" after execution passes the sequence point on line 14? How to ensure this?

According to the object semantics of the C language, detailed in Appendix F:

- line 7 defines an object of type "int" designated by the name "x;"
- line 9 defines an object of type "int*" designed by the name "p;" and
- it also defines the expression "*p" to be a secondary designator for the object designated by "x."

```
1  int offset = 0;
2  int* identity(int* p)
3  { return p + offset; }
4  thread foo(shared int x) { ... }
5
6  void bar(void) {
7    int x = 5;
8    int f, y;
9    int *p = &x;
10   p = identity(p);
11
12   create(f; 0; 1; 1; 0) foo(x);
13   sync(f);
14   y = *p;
15 }
```

Listing G.5: Pointer aliasing a channel endpoint.

From that point onward, C mandates that "x" and "*p" refer to the same object, as well as any expression of the form "*E" where the pointer value E is equal *at run-time* to the value of the expression "&x." Since expressions of the form "*E," when the value of E is not statically known, are always handled as memory loads, this implies that the object designed by both "x" and "*p" must be stored in memory, at an address copied to the pointer object designed by "p." This applies here because the value of the object designed by "p" cannot be statically determined after line 10 (since the global-scope variable "offset" *may* have a non-zero value at run-time, it is not possible to derive statically the certainty that the function named "identity" always returns its argument).

However, when execution reaches the "`create`" construct, the value of the object designed by "x" *must* be present in machine register so it can become the source value for the input communication in the thread program "foo" (section 4.3.3.3) This appears at first sight to conflict with the declaration semantics described above; to resolve this conflict a compiler must duplicate the object into a register upon the "`create`" construct, reserve the register until the "`sync`" construct, and copy back the value from the register to the same memory location immediately after the "`sync`" construct so that the subsequent use of "*p" produces the desired behavior.

This entails extra complexity in the implementation because existing C compilers internally only associate one storage for each object at each point in the control flow graph (either register or memory). Here the memory location must be preserved, alongside the register name used as channel endpoint, along the edges of the control flow at all points between the "`create`" and "`sync`" constructs.

G.2.3 Two-way implicit type conversions

Consider the example program fragment in listing G.6. In this fragment the name "z" designates an object of type "float." However, the thread interface of "foo" expects a channel of type "int." In the machine interface, different register classes are used for integer and floating-point values: the value read by the instance of "foo" *must* be present in an integer register. This appears to conflict with the declared type of "z." To resolve this conflict, a

```
1  thread foo(shared int x) {
2      x = x / 2;
3  }
4  ...
5  {
6      int f;
7      float y, z = 4.5;
8      create(f; 0; 1; 1; 0) foo(z);
9      sync(f);
10     y = z;
11 }
```

Listing G.6: Implicit endpoint type conversion.

```
1  thread foo(shared int a,
2         shared int b, shared int c);
3  thread bar(shared int d,
4         shared int e, shared int f);
5  ...
6  {
7     int x = 1, y = 2, z = 3;
8     int f1, f2;
9     create(f1; 0; 1; 1; 0) foo(x, y, z);
10    create(f2; 0; 1; 1; 0) bar(z, y, x);
11    sync(f1);
12    sync(f2);
13 }
```

Listing G.7: Multiple orderings of the same endpoint names.

compiler must automatically reserve an integer register upon the "`create`" construct and convert the floating-point value from the object designated by "z" to this register prior to the family creation. Then it must also convert back the final value of this register after the "`sync`" construct into the object designated by "z."

This entails extra complexity in the implementation because existing C compilers only introduce automatic type conversions between sequence points for designated objects. Here the conversion must occur before the sequence point that immediately follows "`sync`", but there is no object designator mentioned at point in the program text. To support this a compiler must track internally all objects that need conversion from the point where "`create`" is used until the point "`sync`" is used.

G.2.4 Reuse of "shared" thread arguments

Consider the fragment in listing G.7. This code is valid according all the definitions introduced so far. The difficulty with this example is to determine its semantics. The definitions up to this point open two possible views:

- in the first view, which we shall name "abstruse and broken" (A&B) hereafter, the variables designated by "x," "y," and "z" are established as *semantically synonymous*

with the communication channel endpoints. In this view, the first creation at line 9 has well-defined semantics, and the initial values for "x," "y" and "z," respectively 1, 2 and 3, are used as the source values for the first created family. Then, because the names are synonymous with the channel endpoints, their value becomes undefined immediately after the "`create`" construct: as soon as the first creation occurs, there is no scheduling guarantee and the created thread may concurrently output new values to either "x," "y" or "z" before the second "`create`" construct is reached at line 10 in the creating thread. So the second creation has a race condition. Then another race condition exists at the first synchronization on line 11 since the second family *may* write to its final output "shared" channel before the first family does and terminates. This second race condition persists even if the synchronization order is inverted.

- in the second view, which we shall name "cumbersome and defective" (C&D) hereafter, the variables designated by "x," "y," and "z" have their own storage, are *copied* to channel "buffers" upon family creation, and copied back from the channel buffer to their own storage upon family synchronization. In this view, both creations have well-defined semantics, and the initial values for "x," "y" and "z," are used as source values for all "shared" channels "a" to "f." Then the values of the variables remain unchanged in the creating thread until line 11 is passed, at which point they are updated with whichever values were produced by the first family; then they are updated again at line 12 with the values produced by the second family.

More generally, with the view A&B, the semantics become undefined due to race conditions as soon as a common variable name is used as "shared" thread argument for two families created from the same thread concurrently. This race condition is resolved in view C&D, at the expense of doubling the space requirement and extra copy operations at each creation and synchronization. The extra space requirement is especially troublesome, because it must be allocated from the register name space which is typically small (32/64 registers). Also, these extra costs defeat a primary benefit of the architecture, namely *lean* and *fast* family creation and synchronization.

G.2.5 Summary of the hidden complexity

We do not detail further the implementation complexity, in a compiler, of either of these views, because we eventually side-stepped this issue entirely in the solution presented in chapter 6. However, we can summarize this section as follows: were the proposed language extensions adopted, this examples exposes yet another unforeseen complexity in the design: either broken semantics, or cumbersome extra channel buffers.

G.3 Fundamental problem with the original language proposal

In this section, we prove that the language described in this appendix, in contrast to the one described in chapter 6 and Appendix I, cannot be compiled to a machine interface with "fused" creation such as described in section 4.3.1.2.

G.3.1 Non-composability within statement blocks

Let us consider the program fragment in listing G.8. This fragment is correct according to the original proposed language semantics described above. Its control flow is given in fig. G.1.

```
1  thread foo(shared int x);
2  int test(int x);
3
4  void bar(int N) {
5     int i, x, f;
6     for (i = 0; i < N; ++i) {
7        x = i;
8        create(f; 0; 1; 1; 0) foo(x);
9        if (test(f) == 0) {
10         sync(f);
11         i = x;
12       }
13    }
14 }
```

Listing G.8: Example program fragment using "`create`"

When control enters the body of the function "bar," it reaches a loop. At each iteration, a concurrent work unit is created with one logical thread, given the iteration counter as input value for a "shared" channel. The family identifier value produced by the "`create`" construct is then provided as argument to the function "test," and depending on whether the call to "test" evaluates to 0, the family is synchronized upon before the next iteration starts and the iteration counter reset with the output value of the family's "shared" channel. The important property of this example is that the number of loop iterations, and whether the family is synchronized upon, is decided at run-time: neither the value of "N" nor the return value of "test" can be proven statically. Moreover, it is not possible to reorder the loop body, because there is a data dependency between "`create`" and the call to "test," a control flow dependency between the call to "test' and the use of "`sync`," and a data flow dependency between each use of "`sync`," and the next loop iteration. Finally, if the thread program "foo" is known to be deterministic, and "test" is known to always returns 0 at run-time, all executions of the function "bar" will always be fully determinisitic (in particular without concurrent race conditions) regardless of the choice of semantics opened in Appendix G.2.4.

Although the program fragment has well-defined semantics, it *cannot* be compiled to the target machine interface using "fused" create operations (section 4.3.1.2).

To prove this, we start by reminding ourselves of the semantics of the fused creation: this binds a machine register name from the point of family creation until the family termination *at run-time*. To cause the thread execution to wait until termination of the created family (the effect of running through "`sync`" in the control flow), it *suffices* that execution reaches an operation that uses this register name as operand. Conversely, in order to *avoid* synchronization on termination (the effect of *not* running through "`sync`" in the control flow), it is *necessary* that all operations executed after the "create" instruction avoid using this register name as operand *until the thread terminates*, i.e. here until execution exits the control flow graph of "bar."

Then we consider all potential encodings of basic block BB2 in fig. G.1. For any potential valid encoding E of this basic block, E necessarily contains an occurrence of the "create" instruction, since the "`create`" construct is present in the source language. Let us consider the register name "$R" used as operand to the "create" instruction in E. To support the run-time case where edge E3 is taken and the family termination is not synchronized upon by

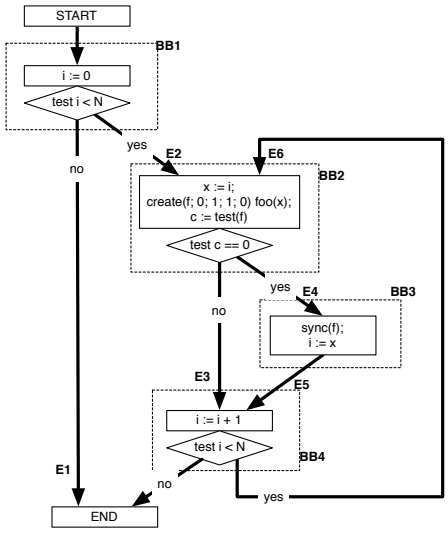


Figure G.1: Control flow graph of the function "bar" in listing G.8

the thread, necessarily *all further basic blocks down the control flow graph must avoid using "$R."* This includes basic block BB4, but also basic block BB2 due to the backward edge E6. Therefore E must not use "$R," which contradicts the definition of "$R." The contradiction entails that there exists no valid encoding of basic block BB2, i.e. that listing G.8 cannot be compiled to the machine interface.

This example is sufficient to prove that the source language cannot be compiled to a machine interface with fused creation, since there exists some valid input programs without a valid target encoding. When we initially proposed this proof, we were told that the example is convoluted, and that the input language definition might still be "good enough" if programs without cycles in the control flow graph can be compiled. Unfortunately, this is not possible either.

To understand why, let us consider the program fragment in listing G.9. This fragment contains a linear sequence of N uses of the "`create`" construct, where N is equal or greater the number of hardware register names M available in the target machine interface (e.g. M = 32, N = 33). Again, this program fragment is valid according to the original proposed language semantics. Let us thus consider any potential encoding E of this fragment. As seen above, the encoding of each "`create`" construct must avoid using any register name used by all previous "`create`" constructs that dominate it in the control flow graph. For the first "`create`" construct, one of M register names is used in E. For the second "`create`" construct, one of $M - 1$ register names is used in E. And so on inductively, until the Mth "`create`" construct, where one in 0 remaining register names is used in E. Since this is not possible, E does not exist, i.e. listing G.9 cannot be compiled.

More generally, any potential compilation scheme for the proposed "`create`" construct, which can be used wherever a C statement can be used, does not compose under the sequential composition of C statements.

```
1  thread foo(void);
2  thread bar(shared int x);
3
4  void baz(void) {
5    int f1, f2, ... f⟨N⟩;
6    int x1, x2, ... x⟨N-1⟩;
7    create(f1;0;1;1;0) foo();
8    x1 = f1;
9    create(f2;0;1;1;0) bar(x1);
10   x2 = f2;
11   ...
12   create(f⟨N-1⟩;0;1;1;0) bar(x⟨N-2⟩);
13   x⟨N-1⟩ = f⟨N-1⟩;
14   create(f⟨N⟩;0;1;1;0) bar(x⟨N-1⟩);
15   sync(f⟨N⟩);
16   if (x⟨N-1⟩) {
17     sync(f1);
18     sync(f2);
19     ...
20     sync(f⟨N-1⟩);
21   }
22 }
```

Listing G.9: Example program fragment using the "`create`" construct.

G.3.2 Soundness argument

When exposed with the argument above, the designers of the original proposal suggested to further restrict usage of the "`create`" construct in specific programs depending on *whether register allocation succeeds for the surrounding thread program*; that is, advertise to users of the compiler technology that the *validity* of source code can only be checked by *actually* trying to process the source code through the *specific implementation* of register allocation in a core compiler technology.

This strategy amounts to defining a programming language that is *unsound*: if two program fragments A and B are valid because they can be compiled to their machine representation $[A]$ and $[B]$, it does not entail that the semantically valid sequential composition "$A;B$" can be compiled to a machine representation $[A;B]$.

Since we should deem soundness a strongly desirable property of any machine interface to a general-purpose processor architecture, an opinion shared throughout our research community, we conclude that the proposed interface language was *inappropriate* for use with the proposed architecture.

G.4 Summary and conclusion

In this appendix, we have thoroughly analyzed prior work on designing a C-based interface to hardware microthreading. Our analysis shows shortcomings of the prior language design, including the impossibility to compile the proposed language constructs to a hardware implementation offering "fused" thread creation as proposed in section 4.3.1.2. This conclusion justifies why we did not reuse and built upon this prior work.

Appendix H

"Quick and dirty" compilation

—Massaging existing technology as a practical approach to code generation

Abstract

This appendix complements chapter 6 and details how we implemented code generation from source code to the new target architecture. To achieve this, we encapsulated an *existing* C compiler *unchanged* between a context-free source-to-source transformer and a post-processor on the assembly source. We then encapsulated the entire compilation pipeline within a new user-facing command, which we call "`slc`," with command-line semantics similar to GNU CC's "`gcc`" driver.

Contents

```
1    .globl foo
2    .ent foo
3    .registers 1 0 3 0 0 0
4  foo:
5    ldpc $l2
6    ldah $l2,0($l2) !gpdisp!1
7    lda $l2,0($l2) !gpdisp!1  # $l2 := GP
8    ldq $l1,a($l2) !literal   # $l1 := &a
9    ldq $l1,0($l1)            # $l1 := a
10   swch
11   s4addq $l0,$l1,$l0        # $l0 := &(a[i])
12   swch
13   stl $g0,0($l0)            # a[i] := x
14   end
15   .end foo
```

Listing H.1: Hand-crafted thread program.

```
1  extern int * a;
2  void foo(int i, int x) {
3    a[i] = x;
4  }
```

Listing H.2: Hand-crafted C code.

H.1 Thread programs and "global" channels

We started by observing that besides the machine interface features directly related to multithreading, the machine code executed by a logical thread is identical to the machine code that would be executed by a procedure on a legacy architecture with the same ISA substrate. We thus conjectured that an existing code generator for the underlying ISA would be able to generate valid code for the "body" of thread programs on our architecture.

To test this conjecture, we considered random C patterns, then hand-crafted their valid representation as a thread program in assembly code. Separately, we compiled the C code with a regular (legacy) C compiler towards the same ISA. Then we compared the results.

As an example, let us consider the assignment "a[i] = x" where "a" is declared in the global scope, "i" is the logical thread index and "x" is a parameter. A valid enclosing thread program in assembly code is expressed in listing H.1. In this example, the ".registers" directive indicates 1 "global" channel endpoint and 3 private (local) registers, as per sections 4.3.3 and 4.8.1. The first instruction loads the program counter in register $l2. The next two instruction load the GP pointer into register $l2 (cf. [Foua] for information about GP-based addressing on the Alpha ISA). The next "ldq" instruction loads the address of pointer "a" using the GP pointer, and the next "ldq" instruction loads "a" itself. Then the array is accessed as expected, using the thread index preloaded in $l0 as per section 4.3.2.3. The "swch" and "end" annotations indicate a fetch switch hint and thread termination, respectively (cf. sections 4.4 and 4.8.1).

We then wrote the C code in listing H.2 and compiled it to Alpha assembly using the GNU C compiler. The result is shown in listing H.3. When comparing this result with

```
1    .globl foo
2    .ent foo
3  foo:
4    ldah $29,0($27)   !gpdisp!1
5    lda $29,0($29)    !gpdisp!1
6    ldq $1,a($29)     !literal
7    ldq $1,0($1)
8    s4addq $16,$1,$1
9    stl $17,0($1)
10   ret $31,($26),1
11   .end foo
```

Listing H.3: Alpha assembly generated using GNU CC.

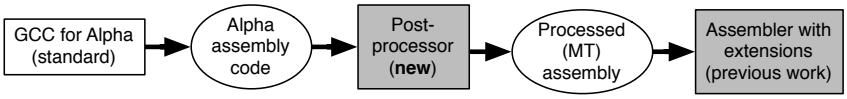


Figure H.1: Position of the assembly post-processor in the tool chain.

listing H.1, the following differences appear: the hand-crafted version contains ".registers" and "ldpc," whereas the compiled version contains "ret" and a different use of register names. Otherwise, the instruction order and data flow are similar (the same in this example).

This similarity of structure persisted across most code fragments we exercised in this way. This allowed us to derive the minimal set of *transformations* needed to obtain the valid microthreaded assembly from the output of GNU CC:

1. insert ".registers" before the start of the function;
2. substitute the register names to match the structure of the thread register window;
3. insert ".ldpc" before the "ldah/lda" pair;
4. remove "ret" at the end, replace with "end" (end thread);
5. add "swch" where appropriate.

The 3 latter steps can be automated on the assembly text without interacting with the originating compiler. We did so by implementing a *post-processor*, inserted between the code generator from GNU CC and the assembler program, as depicted in fig. H.1. To add "swch," we simply performed a definition-use analysis on the data flow of the assembly instruction operands, and placed "swch" after every instruction that *may* consume the output of a long-latency operation (e.g. the first instruction downstream of a memory load, FP operation, etc.)

Inserting ".registers" and translating the register names using a static translation table was possible in the post-processor as well, however this needs to know the desired thread interface. The thread interface is not expressed in the GNU CC output using the source in listing H.2. Also, we needed to ensure that the register name corresponding for the "global" channel endpoint would not be used by the code generator for other computations. Finally, we noticed that the "ret" instruction was often preceded by frame adjustments which are unnecessary in a thread program.

```
1 extern int * a;
2 sl_def(foo, int, x) {
3   sl_index(i);
4   a[i] = x;
5 }
6 sl_enddef
```

Listing H.4: C code with strategically placed macro uses.

```
1 #define sl_index(IdxName)  long IdxName = __mt_index
2 #define sl_def(FunName, Arg1Type, Arg1Name)              \
3   void FunName(void) {                                   \
4     register long __mt_index __asm__("$28");             \
5     register Arg1Type Arg1Name __asm__("$1");            \
6     __asm__ __volatile__(".registers 1 0 0 0 0 0");
7 #define sl_enddef }
```

Listing H.5: Macro definitions for thread programs.

We automated the translation as follows. First we re-expressed the code from listing H.2 as depicted in listing H.4, hiding the function header and footer behind specially-named pre-processor macros. Then we implemented a helper file `sl_support.h` with the definitions from listing H.5. (Note that we use names prefixed with a double underscore "`__`" to void conflicts with existing identifiers in C code, cf. [II99, 7.1.3§1] and [II11b, 7.1.3].)

Then we needed to manage register renaming.

By investigating the GNU CC code generation back-end, we discovered that register allocation is performed in two passes: first all candidate variables are assigned to pseudo-registers, trying to minimize the number of pseudo-registers used, then the pseudo-registers are assigned to machine register names from a pool, *in a fixed order*. In other words, the implementation is such that a function that needs N different registers will always use the N first candidates in the allocation order. For this ISA target, the allocation order is 1, 2, 3, 4, 5, 6, 7, 8, 22, 23, 24, 25, 28, 0, 21, 20, 19, 18, 17, 16, 27, 9, 10, 11, 12, 13, 14, 26, 15 (`REG_ALLOC_ORDER` in `gcc/config/alpha/alpha.h`).

We used this newly gained knowledge as follows. First we excluded the first 12 registers of the allocation order using the command-line option "`-ffixed-$`*reg*" [Frec], in order to "make room" for inter-thread channel endpoints. We also compiled with "`-include sl_support.h`" to automatically prepend listing H.5 to the pre-processing unit:

```
alpha-linux-gnu-gcc -ffixed-$1 -ffixed-$2 \
   -ffixed-$3 -ffixed-$4 -ffixed-$5 \
   -ffixed-$6 -ffixed-$7 -ffixed-$8 \
   -ffixed-$22 -ffixed-$23 -ffixed-$24 \
   -ffixed-$25 \
   -include sl_support.h  -O2 test.c -S
```

The GNU CC output with these options is given in listing H.6. As desired, the name "$1" is only used in its role as a channel endpoint. The temporary variables are assigned

```
1    .globl foo
2    .ent foo
3  foo:
4    ldah $29,0($27) !gpdisp!1
5    lda $29,0($29)  !gpdisp!1
6    .registers 1 0 0 0 0 0
7    ldq $0,a($29)   !literal
8    ldq $0,0($0)
9    s4addq $28,$0,$28
10   stl $1,0($28)
11   ret $31,($26),1
12   .end foo
```

Listing H.6: Externally instrumented Alpha assembly.

Register name used by GNU CC	$1	$2	$3	$4	$5	$6
Standard alias/role	$t0	$t1	$t2	$t3	$t4	$t5
Substituted name	$g0/$d5	$g1/$s5	$g2/$d4	$g3/$s4	$g4/$d3	$g5/$s3
Register name used by GNU CC	$7	$8	$22	$23	$24	$25
Standard alias/role	$t6	$t7	$t8	$t9	$t10	$t11
Substituted name	$g6/$d2	$g7/$s2	$g8/$d1	$g9/$s1	$g10/$d0	$g11/$s0
Register name used by GNU CC	$28	$0	$21	$20	$19	$18
Standard alias/role	$at	$rv	$a5	$a4	$a3	$a2
Substituted name	$l0	$l1	$l2	$l3	$l4	$l5
Register name used by GNU CC	$17	$16	$27	$9	$10	$11
Standard alias/role	$a1	$a0	$pv	$s0	$s1	$s2
Substituted name	$l6	$l7	$l14	$l8	$l9	$l10
Register name used by GNU CC	$12	$13	$14	$26	$15	$29
Standard alias/role	$s3	$s4	$s5	$ra	$fp	$gp
Substituted name	$l11	$l12	$l13	$l15	$l16	$l17
Register name used by GNU CC	$30	$31				
Standard alias/role	$sp	$31				
Substituted name	$l18	$31				

Table H.1: Substitution table for register names.

to register names past the reserved range ("$28", "$0"). The ".registers" directive is present, albeit not at the right position.

To refine the output, we then added the following steps in the post-processor:

1. substitute the register names using table H.1;
2. find the highest local register name *actually* used;
3. pull the inserted ".registers" directive at the top of the function definition and replace its 3rd parameter by the index of the highest local register name.

The result of these steps on listing H.6 is given in listing H.7. This is a valid thread program, which can be run on the new architecture.

However, some extra effort is required. As we can see in the edited assembly, as a special case in the allocation strategy outlined above, the special GP pointer is always assigned to "$29." The result of the substitution thus always uses the name "$l17" if the original code needed GP, even though the thread program only needs 3 private registers for the computation. This is unsatisfactory, because the machine interface strongly suggests to

```
1    .globl foo
2    .ent foo
3    .registers 1 0 18 0 0 0
4  foo:
5    ldpc $l17
6    ldah $l17, 0($l17) !gpdisp!1
7    lda $l17, 0($l17) !gpdisp!1
8    ldq $l1,a($l17)  !literal
9    swch
10   ldq $l1,0($l1)
11   s4addq $l0,$l1,$l0
12   swch
13   stl $g0,0($l0)
14   end
```

Listing H.7: Automatically edited Alpha assembly.

```
1    .globl foo
2    .ent foo
3    .registers 1 0 3 0 0 0
4  foo:
5    ldpc $l2
6    ldah $l2, 0($l2) !gpdisp!1
7    lda $l2, 0($l2) !gpdisp!1
8    ldq $l1,a($l2)  !literal
9    swch
10   ldq $l1,0($l1)
11   s4addq $l0,$l1,$l0
12   swch
13   stl $g0,0($l0)
14   end
```

Listing H.8: Edited Alpha assembly with fewer used local registers.

reduce the number of effective private registers required by a thread program (sections 3.3.3 and 4.3.3). To address this, we add a "*compression*" algorithm to the post-processor. This algorithm iterates through all used local register names, and for each name it tries to find another name with a lower index which is not yet used. If such a name is found, the name is substituted throughout. This compression occurs before the register count in ".registers" is computed. With this algorithm, we obtain the final output in listing H.8. This is identical to the hand-crafted version from listing H.1: we succesfully compiled a thread program, without changes to the substrate code generator in GNU CC.

H.2 Multiple channel endpoints

The previous mechanism relies on a C pre-processor macro to inject the desired thread interface, expressed by macro arguments, in the source code as variable declarations and the ".registers" directive. Since the C pre-processor cannot compute functions of the number of

```
#define sl_def1(FunName, Type1, Name1)
  void FunName(void) { \
    register long __mt_index __asm__("$28"); \
    register Type1 Name1 __asm__("$1"); \
    __asm__ __volatile__(".registers 1 0 0 0 0");
#define sl_def2(FunName, T1, N1, T2, N2) \
  void FunName(void) { \
    register long __mt_index __asm__("$28"); \
    register T1 N1 __asm__("$1"); \
    register T2 N2 __asm__("$2"); \
    __asm__ __volatile__(".registers 2 0 0 0 0 0");
#define sl_def2(FunName, T1, N1, T2, N2, T3, N3) \
  void FunName(void) { \
    register long __mt_index __asm__("$28"); \
    register T1 N1 __asm__("$1"); \
    register T2 N2 __asm__("$2"); \
    register T3 N3 __asm__("$3"); \
    __asm__ __volatile__(".registers 3 0 0 0 0 0");
/* ... and so on. */
```

Listing H.9: Macro definitions necessary for different interface arities.

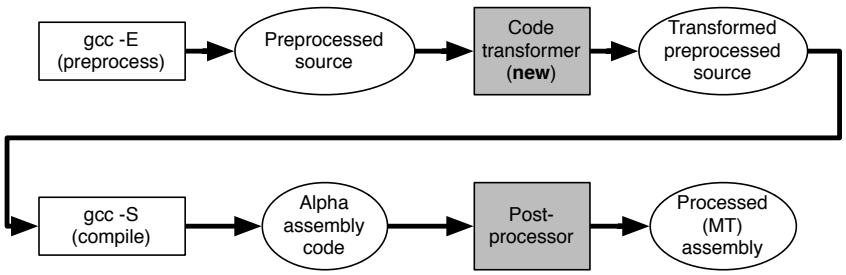


Figure H.2: Position of the code transformer in the tool chain.

macro arguments, this approach would require multiple macro definitions to support more than one channel endpoint, as in listing H.9.

While this approach is conceptually sound, it is technically impractical: for any set of macros the effective number of thread channels supported is limited by the largest macro definition. We considered instead that a dynamically computed expansion of the macro would be a more robust, future-proof approach. To achieve this, we inserted a call to M4 [KR77] as a *code transformer* between the *pre-processing* and *translation* phases of the C compiler, as depicted in fig. H.2, and we defined our "`sl_`" macros using M4.

While implementing the code transformer, we found that the implementation would be simplified if there was a syntactic unit around the pair "(type, name)" for each thread channel. We did so by extending the syntax definitions with a new form "`sl_parm(`*Type*`,` *Name*`)`."

With this setup, we are able to compile thread programs with multiple channel endpoints, e.g. listing H.10, to valid assembly output, e.g. listing H.11 (the comments on the right side of the generated assembly have been manually inserted to explain the result).

Side note H.1: About the avoidance of a C syntax parser.

Although we could have reused an existing comprehensive code transformation infrastructure able to understand the C syntax and semantics, we deliberately avoided doing so for the following practical reasons:

- overhead: the entry cost of learning about and mastering the existing tool seemed large given the relative simplicity of our approach;
- test cycles: the existing tools we found (e.g. CIL [NMRW02]) required separate build and execution steps to use; we thus deemed them inappropriate for the agile development style we wished to use;
- simplicity: since the syntax substitutions could be implemented in a context-free manner, any error could be easily traced to its origin by looking at the result of the substitution; with a structured tool instead, one would need to scan through a graph or tree of abstract representation to find the origin of errors.

As the story goes, we ended up implementing most the code transformer with Python [Foub] (instead of M4) to simplify the expression of the transformations. However, throughout our research the tool could perform context-free syntax substitutions. Our major goal, which was to avoid knowledge of the potential syntax extensions of the underlying C compiler, while keeping the ability to propagate them fully from input to output, was achieved this way.

Side note H.2: Preservation of line number markers in M4.

When adding M4 to the processing chain, we had to modify M4 to understand the line number markers added during the initial C pre-processing step ("`#line` *N*"). This is necessary because macro expansion in M4 may add or remove lines from the pre-processed text. Without extra support from M4, a discrepancy could appear between the line markers originally inserted by the C pre-processor and the source code after processing by M4, which would in turn render any subsequent error messages during compilation meaningless.
We performed this change in GNU M4. The change was minimal (less than 20 lines of code modified); it was submitted to the GNU M4 developers and is pending acceptance in the main GNU M4 distribution.

```
1 sl_def(scal, sl_parm(int*, b),
2         sl_parm(const int*, a),
3         sl_parm(int, c)) {
4   sl_index(i);
5   b[i] = a[i] * c;
6 }
7 sl_enddef
```

Listing H.10: Code using multiple channel endpoints.

```
1           .globl scal
2           .ent scal
3           .registers 3 0 2 0 0 0  # 3 chans, 2 local regs
4 scal:
5           s4addq $l0,0,$l0 # $l0 := index * 4
6           addq $g1,$l0,$l1 # $l1 := a + index*4 (= &a[i])
7           addq $g0,$l0,$l0 # $l0 := b + index*4 (= &b[i])
8           ldl $l1,0($l1)   # $l1 := a[i]
9           mull $l1,$g2,$l1 # $l1 := a[i] * c
10          swch
11          stl $l1,0($l0)   # store int to b[i]
12          swch
13          end
14          .end scal
```

Listing H.11: Generated assembly source for the "scal" thread program.

H.3 "Shared" channel endpoints

To implement "shared" channel endpoints (section 4.3.3.3), we could use register names from the same reserved block of 12 names. However, a new difficulty arised: there are now two endpoints per channel, one "incoming" and "outgoing." Meanwhile, we wanted to keep the general look and feel of the original language proposal, and allow users to define a single name to designate both. We did this as follows. First we defined a new form "`sl_shparm(`*Type*`,`*Name*`)`" which populates the 2nd parameter to ".registers" and generates two additional declarations at each use in the C function header, for example:

```
register Type __mti_Name __asm__("$24");
register Type __mto_Name __asm__("$25");
```

Then we defined two other forms as follows:

```
sl_getp(Name)
→ (__mti_Name)
sl_setp(Name,Value)
→ do { __mto_Name = (Value); } while(0);
```

(The use of "do...while(0)" ensures that the new construct can only be used in the syntax where a C statement is expected.)

While experimeting with this new form, we ran into two issues, related to the ability of the substrate GNU CC code generator to track aliases through assignments, which is otherwise a desirable optimization. The first issue is illustrated by the following sequence:

```
int t,u;
t = sl_getp(s);
sl_setp(s, t+1);
int u = t * t;
```

Without further attention, this would be translated to:

```
lda $l0, 1($31)     # $l0 := 1
addl $d0, $l0, $s0 # $s0 := $d0 + 1
mull $d0, $d0, $l0 # $l0 := $d0 * $d0
```

In other words, the code generator tracks that "t" is an alias for "`__mti_`*Name*" and reuses the same register name beyond "`sl_setp`." This is incorrect in our setting, as explained in side note G.2. To address this, it suffices to indicate that any use of "`sl_setp`" also clobbers "`__mti_`*Name*," to force "t" to alias a copy beyond that point. We do this as follows:

```
sl_setp(Name,Value)
→ do {
    __mto_Name = (Value);
    __asm__ __volatile__("#nothing"
              : "=r"(__mti_Name)
              : "r"(__mto_Name));
} while(0);
```

As per [Frea] this syntax ensures that any prior alias to "`__mti_`*Name*" cannot use the original register name any more.

Then we experienced another issue: some C expressions would cause the target of an assignment to be assigned multiple times. For example:

```
sl_setp(s, x ? (x - 1) : 0);
```

would be compiled to:

```
subl $10,1,$s0    # $s0 := $10 - 1
cmoveq $10,0,$s0 # if $10=0 then $s0:=0
```

This is incorrect because each output to an outgoing channel is a synchronizing event and only the final result should be visible by the sibling thread. We addressed this by extending the definition of "`sl_setp`" as follows:

```
sl_setp(Name,Value)
→ do {
    __typeof__(__mto_Name) __tmp_set = (Value);
    __asm__ __volatile__("mov %4,%0"
              : "=r" (__mto_Name),
                "=r" (__mti_Name)
              : "0" (__mto_Name),
                "1" (__mti_Name),
                "r" (__tmp_set));
} while(0);
```

With this construct, the value to output to the outgoing "shared" endpoint is always constructed in a local synchronizer before the final "mov" instruction effects the output. Note how we use "typeof" to define a temporary variable of the right type. Although we could have defined a parameter type lookup table in the code transformer, which would "remember" throughout a thread program body the type of each channel endpoint, this would fail to handle the following source:

```
typedef long mytype;
sl_def(foo, sl_shparm(mytype, x)) {
    typedef char mytype[1000];
    sl_setp(x, sl_getp(x)+1);
}
```

In this example the name "mytype" is changed to designate a different type within the function body, and thus after the function header. This new definition would be invisible to the code transformer because it appears outside of one of our constructs. If we would reuse the name "mytype" textually in the expansion of "`sl_setp`," the code would become invalid. With "typeof" we ensure that the original definition type is used.

Finally, for symmetry, we then renamed the previous "`sl_parm`" construct for "global" channels to "`sl_glparm`" and we ensured that it declares variables of the form "`__mti_`*Name*" so that "`sl_getp`" can be used to access them.

With this in place, we are able to compile listing H.12 to listing H.13, which is a valid thread program for the new architecture.

H.4 Bulk creation

H.4.1 Code generation for "fused" creation

We cover first code generation for "fused" creation styles (section 4.3.1.2), as this was historically the first interface available.

```
sl_def(innerprod, sl_glparm(int*,a),
       sl_glparm(int*,b),
       sl_shparm(int,sum))
{
  int* a = sl_getp(a);
  int* b = sl_getp(b);
  sl_index(i);
  sl_setp(sum, a[i] * b[i] + sl_getp(sum));
}
sl_enddef
```

Listing H.12: Source code for the "innerprod" thread program.

```
  .globl innerprod
  .ent innerprod
  .registers 2 1 2 0 0 0
innerprod:
 s4addq $l0,0,$l0 # $l0:=i * 4
 addq $g1,$l0,$l1 # $l1:=&b[i]
 swch
 addq $g0,$l0,$l0 # $l0:=&a[i]
 swch
 ldl $l1,0($l1)   # $l1:=b[i]
 ldl $l0,0($l0)   # $l0:=a[i]
 swch
 mull $l1,$l0,$l0 # $l0:=a[i]*b[i]
 swch
 addl $l0,$d0,$l0 # $l0:=a[i]*b[i]+$d0
 swch
 mov $l0,$s0
 end
```

Listing H.13: Generated assembly for the "innerprod" thread program.

As described in section 4.3.1, bulk creation is controlled at the machine interface via a sequence of "allocate," "set" configuration operation and one of the "create" operations. To follow the look and feel of the original language proposal, we encapsulated this sequence behind a single construct, given in listing H.14.

We use the register constraint "rI" because the configuration instructions accept both an immediate constant or a register, and the code generator will choose the form that entails the least number of instructions. We use a counter in the code transformer to auto-generate a different value for N at every occurrence of "`sl_createsync`," to avoid duplicate declarations of "`__mt_fid`."

With this in place, we can compile listing H.15 to listing H.16.

Also, we use a condition in the code transformer to avoid emitting one of the configuration instruction if the corresponding parameter is omitted in the form "`sl_createsync`." For example, our construct allows us to compile listing H.17 to listing H.18.

The next step concerns setting up communication endpoints with the created family. In the hardware implementations we have used, fused creation is always used with "hanging"

```
1  sl_createsync(Start, Limit, Step, Block, Fun)
2  →
3  long __mt_fidN;
4  __asm__ __volatile__("allocate␣%0" : "=r"(__mt_fidN));
5  __asm__("setstart␣%0,␣%2"
6    : "=r"(__mt_fidN) : "0"(__mt_fidN), "rI"(Start));
7  __asm__("setlimit␣%0,␣%2"
8    : "=r"(__mt_fidN) : "0"(__mt_fidN), "rI"(Limit));
9  __asm__("setstep␣%0,␣%2"
10   : "=r"(__mt_fidN) : "0"(__mt_fidN), "rI"(Step));
11 __asm__("setblock␣%0,␣%2"
12   : "=r"(__mt_fidN) : "0"(__mt_fidN), "rI"(Block));
13 __asm__ __volatile__("crei␣%0,␣0(%1);␣mov␣%0,␣$31"
14   : : "r"(__mt_fidN), "r"(Fun))
```

Listing H.14: Prototype "create" construct.

```
1  sl_def(foo) { } sl_enddef
2  sl_def(bar) {
3    sl_createsync(1, 2, 3, 4, foo);
4  } sl_enddef
```

Listing H.15: Example use of "createsync."

synchronizer mappings (section 4.3.3.3). As per section 4.8.2, with such mappings the "setregs" instruction must specify static offsets in the creating thread's register window where the endpoints must be set up. Also, these endpoints must contain the source values at the point the "create" instruction is executed. Finally, the endpoints are set up by using contiguous register names from the first offset indicated by "setregs."

To implement this, we force the use of specific registers to pass arguments, as detailed in listing H.19. First we extend the interface of the "`sl_createsync`" form to accept a variable number of extra constructs of the form "`sl_glarg(`*Type*`, `*Name*`, `*Value*`)`." When G such constructs are provided, we then declare G local variables named accordingly, placing them in explicit registers. We choose the explicit register names by applying the *reverse substitution* from table H.1 to the new names "$l0...$l($G-1$)." This produces the G first names from the sequence \$28, \$0, \$21, ... \$26, with the guarantee that the forward substitution will produce contiguous register names during post-processing.

We then expand the context allocation and family configuration as previously.

Then we assign the values specified in the "`sl_glarg`" forms to the new local variables. We place these assignment after the family configuration, instead of directly as initializers in the variable declarations, in order to preserve the order of side effects in the expressions passed to "`sl_createsync`."

Finally, during creation, we force the new variables to be available in their respective registers at the point the "create" instruction is emitted. We construct the meta-syntactical variables "%N" in the assembly pattern based on the number G of arguments that follow.

For "shared" channels, we extend as detailed in listing H.20. This is a straightforward extension of listing H.19, using the next S register names in the creating thread's register

```
    .globl foo
    .ent foo
    .registers 0 0 0 0 0 0
foo:
    nop
    end
    .end foo
    .globl bar
    .ent bar
    .registers 0 0 3 0 0 0
bar:
    ldpc $l2
    ldah $l2, 0($l2) !gpdisp!1
    lda $l2, 0($l2) !gpdisp!1    # $l2 := GP
    allocate $l0                 # $l0 := context id
    ldq $l1, bar($l2) !literal   # $l1 := &bar
    setstart $l0, 1              # set start := 1
    swch
    setlimit $l0, 2              # set limit := 2
    setstep $l0, 3               # set step := 3
    setblock $l0, 4              # set block := 4
    crei $l0, 0($l1)             # create, $l0 := future
    swch
    mov $l0, $31                 # wait on $l0 (sync)
    end
    .end bar
```

Listing H.16: Generated asembly from listing H.15.

```
sl_def(foo,sl_glparm(int,x)) {
    sl_createsync( , sl_getp(x), , , bar);
}
```

Listing H.17: Example use of "createsync" with omitted parameters.

```
    allocate $l0                 # $l0 := context id
    ldq $l1, bar($l2) !literal   # $l1 := &bar
    setlimit $l0, $g0            # limit := 1st th. parm
    swch
    crei $l0, 0($l1)             # create, $l0 := future
    swch
    mov $l0, $31                 # wait on $l0 (sync)
```

Listing H.18: Generated assembly for listing H.17.

```
1  sl_createsync(... Fun,
2                sl_glarg(T1, N1, Val1), ... sl_glarg(TG, NG, ValG))
3  →
4   long __mt_fidN;
5   T1 __mta_N1 __asm__("$⟪RevSubst($l0)⟫");
6   ...
7   TG __mta_NG __asm__("$⟪RevSubst($l(G-1))⟫")
8   /* ... allocate and configure happen here ... */
9   __mta_N1 = (Val1);
10  ...
11  __mta_NG = (ValG);
12  __asm__ ("setregs␣%0,␣0,␣0,␣0"
13             : "=r"(__mt_fidN) : "0"(__mt_fidN));
14  __asm__ __volatile__("crei␣%⟪2G⟫, 0(%⟪2G+1⟫)"
15             "\n\tmov␣%⟪2G⟫, $31"
16           : "=r"(__mta_N1), ... "=r"(__mta_NG),
17           : "r"(__mta_N1), ... "r"(__mta_NG),
18             "r"(__mt_fidN), "r"(Fun))
```

Listing H.19: Support for "global" channel endpoints in "createsync."

```
1  sl_createsync(... Fun,
2                sl_sharg(T1, N1, Val1), ... sl_sharg(TS, NS, ValS))
3  →
4   /* declaration for G variables here... */
5   T1 __mta_N1 __asm__("$⟪RevSubst($lG)⟫");
6   ...
7   TS __mta_NS __asm__("$⟪RevSubst($lG+S-1)⟫")
8   /* ... allocate and configure happen here ... */
9   /* ... also assignments to 1st G channel endpoints */
10  __mta_N1 = (Val1);
11  ...
12  __mta_NS = (ValS);
13  __asm__ ("setregs␣%0,␣0,␣⟪G⟫, 0, 0"
14             : "=r"(__mt_fidN) : "0"(__mt_fidN));
15  __asm__ __volatile__("crei␣%⟪2(G+S)⟫, 0(%⟪2(G+S)+1⟫)"
16             "\n\tmov␣%⟪2(G+S)⟫, $31"
17           : /* G arguments here... */,
18             "=r"(__mta_N1), ... "=r"(__mta_NS),
19           : /* G arguments here... */,
20             "r"(__mta_N1), ... "r"(__mta_NS),
21             "r"(__mt_fidN), "r"(Fun))
```

Listing H.20: Extension of listing H.19 for "shared" channels.

```
#define N 100
int A[N];
int B[N];

sl_def(innerprod,
       sl_glparm(int*, a),
       sl_glparm(int*, b),
       sl_shparm(int, sum)) {
  int* a = sl_getp(a);
  int* b = sl_getp(b);
  sl_index(i);
  sl_setp(sum,
          a[i] * b[i] + sl_getp(sum));
} sl_enddef

sl_def(kernel, sl_shparm(int, res)) {
  /* we pass A and B via thread
  channels to avoid extra instruction
  to load the array address in each
  thread of the family. */
  sl_createsync( , N, , , innerprod,
                 sl_glarg(int*,,A),
                 sl_glarg(int*,,B),
                 sl_sharg(int,sum,0));
  sl_setp(res, sl_geta(sum));
} sl_enddef
```

Listing H.21: Example vector-vector product kernel.

window for the endpoints of "shared" channels. We also add a new form "`sl_geta(`*Name*`)`" to retrieve the final value produced by the last thread in the family, as follows:

```
sl_geta(Name) → __mta_Name
```

With these constructs in place, we are able to compile listing H.21 to listing H.22. This example uses the SPARC ISA, as opposed to the Alpha ISA used in previous examples, to illustrate the generality of the approach; it is also one of the benchmarks that was eventually successfully run on the UTLEON3 prototype on FPGA. This implementation uses two machine instructions "setthread/create" instead of only one "crei," but the semantics of the machine interface are otherwise identical.

H.4.2 Code generation with "detached" creation

We cover here the "detached" creation style from section 4.3.1.2.

Here code generation is implified, because the source values of the thread arguments do not need to be placed in specific (contiguous) registers any more. Instead, the "`sl_`×`arg`" constructs now simply expands to a use of the "puts" or "putg" operations for explicit communication of source channel values.

For bulk synchronization, we then expand to an explicit use of the new "sync" and "release" operations. The final values of "shared" communication channels are retrieved into local variables via "gets" between "sync" and "release."

```
1     .global innerprod
2     .type   innerprod, #function
3     .registers      2 1 3 0 0 0
4  innerprod:
5     sll %tl0, 2, %tl1       ! %tl1 := i*4
6     ld [%tg1+%tl1], %tl2    ! %tl2 := b[i]
7     ld [%tg0+%tl1], %tl0    ! %tl0 := a[i]
8     smul %tl2, %tl0, %tl0   ! %tl0 := a[i]*b[i]
9     swch
10    add %tl0, %td0, %tl0    ! %tl0 := a[i]*b[i]+sum
11    swch
12    mov %tl0, %ts0          ! %ts0 := a[i]*b[i]+sum
13    end
14    .size innerprod, .-innerprod
15    .global kernel
16    .type   kernel, #function
17    .registers      0 1 6 0 0 0
18 kernel:
19    allocate %tl3           ! %tl3 := context id
20    sethi %hi(A), %tl0
21    or %tl0, %lo(A), %tl0   ! %tl0 := &A[0] (1st gl.)
22    sethi %hi(B), %tl1
23    or %tl1, %lo(B), %tl1   ! %tl1 := &B[1] (2nd gl.)
24    mov 0, %tl2             ! %tl2 := 0 (1st sh.)
25    setlimit %tl3, 100      ! set limit := 100
26    swch
27    sethi %hi(innerprod), %tl5
28    or %tl5, %lo(innerprod), %tl5  ! %tl5 := &innerprod
29    setthread %tl3, %tl5    ! set PC := &innerprod
30    create %tl3, %tl3       ! create, %tl3 := future
31    swch
32    mov %tl3, %tl3          ! wait on $tl3 (sync)
33    swch
34    mov %tl2, %ts0          ! %ts0 := last sh. output
35    end
36    .size kernel, .-kernel
```

Listing H.22: Generated code for listing H.21, using the "fused" creation interface.

Detached creation is used in the hardware implementations together with the "separated" style of synchronizer mappings (section 4.3.3.3), so the restriction described in side note G.2 does not apply any more, which allows us to simplify the definition of "`sl_setp`" from Appendix H.3.

With this scheme in place, the translation of listing H.21 yields the code in listing H.23.

H.5 Sequential schedule

As explained in section 6.2.4, our technical approach must enable a sequential schedule of any construct expressing concurrency. With the constructs introduced so far, we build an equivalent sequential schedule as follows:

- thread program are replaced by C functions, taking a logical index value as 1st function parameter;
- family creation via "`sl_createsync`" is replaced by a loop;
- "global" channel endpoints are replaced by pass-by-value function parameters;
- "shared" channel endpoints are replaced by single pass-by-reference (pointer) function parameters.

The corresponding syntax expansions are provided in listing H.24. The pre-processor must keep an environment upon encountering "`sl_def`" in order to select the expansion of "`sl_getp`" in the function body (either direct or via pointer indirection); this is possible since we can implement data structures in the code transformer. As previously, the variable declarations at each use of "`sl_createsync`" must be disambiguated using a counter N increased at each occurrence of the construct.

H.6 Floating-point channels

Support for floating-point required extra attention because the register names for floating-point values and operations are different from integer register names.

While it was trivial to reuse the same principle for reserving register names away from register allocation, and establishing a substitution table based on the register allocation order in GNU CC (table H.2), integration in the language constructs mentioned so far ran into an obstacle: it is not possible to infer the *actual type* of a channel argument/parameter for the type name provided in the construct.

The specific question to be answered is: given an occurrence of "`sl_glparm(`*Type*`,` *Name*`)`" (or "`sl_shparm`"), should the expanded text use an integer or floating-point register name? The naive approach suggests looking at the syntactic form of the *Type* textual parameter; however this is inappropriate: if a C *typedef name* is provided ([II99, 6.7.7], [II11b, 6.7.8]), the actual type cannot be infered without looking at the semantic context in the program. Since we wanted to avoid interfering with (or reimplementing) a C compiler front-end, we decided to sidestep the issue entirely and shadow all the relevant constructs with a floating-point variant: "`sl_glfparm`" next to "`sl_glparm`," "`sl_shfparm`" next to "`sl_shparm`," and so on. The other constructs could remain unchanged.

With this extra support in place we were able to successfuly compile code with floating-point channel types.

```
1     .global innerprod
2     .type   innerprod, #function
3     .registers      2 1 3 0 0 0
4  innerprod:
5     sll %tl0, 2, %tl1       ! %tl1 := i*4
6     ld [%tg1+%tl1], %tl2    ! %tl2 := b[i]
7     ld [%tg0+%tl1], %tl0    ! %tl0 := a[i]
8     smul %tl2, %tl0, %tl0   ! %tl0 := a[i]*b[i]
9     swch
10    add %tl0, %td0, %tl0    ! %tl0 := a[i]*b[i]+sum
11    swch
12    mov %tl0, %ts0          ! %ts0 := a[i]*b[i]+sum
13    end
14    .size innerprod, .-innerprod
15    .global kernel
16    .type   kernel, #function
17    .registers      0 1 6 0 0 0
18 kernel:
19    mov 0, %tl1             ! %tl1 := 0
20    allocates %tl0          ! %tl3 := context id
21    sethi %hi(innerprod), %tl2
22    or %tl2, %lo(innerprod), %tl2 ! %tl2 := &innerprod
23    setlimit %tl0, 100      ! set limit := 100
24    swch
25    crei %tl2, %tl0         ! create, %tl0 := acknowledgement
26    sethi  %hi(A), %tl2
27    or  %tl2, %lo(A), %tl2
28    putg %tl2, %tl0, 0      ! send &A to ``global'' 0
29    sethi  %hi(B), %tl2
30    or  %tl2, %lo(B), %tl2
31    putg %tl2, %tl0, 1      ! send &B to ``global'' 1
32    puts %tl1, %tl0, 0      ! send 0 to first ``shared'' 0
33    sync %tl0, %tl1         ! sync, %tl1 := future
34    mov %tl1, %tl1          ! wait on %tl1
35    swch
36    gets %tl0, 0, %tl1      ! retrieve last ``shared'' 0
37    mov %tl1, %tl1          ! wait
38    swch
39    release %tl0            ! release context
40    mov %tl1, %ts0          ! propagate to next thread
41    end
42    .size kernel, .-kernel
```

Listing H.23: Generated code for listing H.21, using the "detached" creation interface.

```
1  sl_def(Name, sl_glparm(T_G,N_G),... sl_shparm(T_S,N_S),...)
2  →
3  void Name(long __mti, T_G N_G, ... T_S *N_G, ...)
4
5  sl_index(Idx) → long Idx = __mti
6
7  sl_getp(N_G) → N_G
8  sl_getp(N_S) → (*N_S)
9
10 sl_setp(N_S, Value) → do { (*N_S) = (Value); } while(0)
11
12 sl_createsync(Start, Limit, Step, Block, Fun,
13               sl_glarg(T_G,N_G,V_G), ...
14               sl_sharg(T_S,N_S,V_S), ...)
15 →
16 long __mtiN, __mtbN = (Start), __mtlN = (Limit), __mtsN = (Step);
17 T_G N_G = (V_G); ...
18 T_S N_S = (V_S); ...
19 for (__mtiN = __mtbN; __mtiN < __mtlN; __mtiN += __mtsN)
20     Fun(__mtiN, N_G, ... &N_S, ...)
21
22 sl_geta(N_S) → N_S
```

Listing H.24: Syntax expansions for a sequential schedule.

Legacy register name	$f10	$f11	$f12	$f13	$f14	$f15
Standard alias/role	$ft0	$ft1	$ft2	$ft3	$ft4	$ft5
Substituted name	$gf0/$df5	$gf1/$sf5	$gf2/$df4	$gf3/$sf4	$gf4/$df3	$gf5/$sf3
Legacy register name	$f22	$f23	$f24	$f25	$f26	$f27
Standard alias/role	$ft6	$ft7	$ft8	$ft9	$ft10	$ft11
Substituted name	$gf6/$df2	$gf7/$sf2	$gf8/$df1	$gf9/$sf1	$gf10/$df0	$gf11/$sf0
Legacy register name	$f28	$f29	$f30	$f0	$f1	$f21
Standard alias/role	$ft12	$ft13	$ft14	$frv	$frv2	$fa5
Substituted name	$lf0	$lf1	$lf2	$lf3	$lf4	$lf5
Legacy register name	$f20	$f19	$f18	$f17	$f16	$f2
Standard alias/role	$fa4	$fa3	$fa2	$fa1	$fa0	$fs0
Substituted name	$lf6	$lf7	$lf14	$lf8	$lf9	$lf10
Legacy register name	$f3	$f4	$f5	$f6	$f7	$f8
Standard alias/role	$fs1	$fs2	$fs3	$fs4	$fs5	$fs6
Substituted name	$lf11	$lf12	$lf13	$lf15	$lf16	$lf17
Legacy register name	$f9	$f31				
Standard alias/role	$fs7	$f31				
Substituted name	$lf18	$f31				

Table H.2: Substitution table for FP register names.

H.7 Forward declarations, separate compilation and visibility

Another requirement was support for separate compilation. Provision for this in the C language exists through the notion of visibility, controlled via the keywords "extern" and "static" (or their absence), and forward declarations. Like thread functions and objects, we must allow the source code to control the visibility of thread programs and express forward declarations.

We enabled this in two steps:

1. we added an additional positional parameter to the "`sl_def`" construct where the word "static" can be expressed optionally, in which case it is injected accordingly in the expansion. The new form then becomes "`sl_def(`$FunName$`,` $Visibility_{opt}$`,` $Channels\ldots_{opt}$`)`";
2. we added a new construct "`sl_decl`" which accepts the same positional parameters as "`sl_def`" and expands to a declaration (without a body).

Later on, we found out that we needed a different textual expansion for declarations of thread programs, and declarations of pointers to thread programs. We introduced an extra form "`sl_decl_fptr`" for this purpose, again with the same positional parameters as "`sl_def`." Although there is no additional associated conceptual difficulty, we mention it here for completeness.

H.8 Extending bulk creation

At a language level, two features outside of the scope of this appendix are also supported with the "`sl_createsync`" construct: the specification of a *place identifier*, which is a named processing resource where the work should be executed (cf. chapter 11 and Appendix E), and extra *creation specifiers* for controlling non-function aspects of code generation (chapter 10). The final syntax interface settled upon was "(, $Place_{opt}$, $Start_{opt}$, $Limit_{opt}$, $Step_{opt}$, $Block_{opt}$, $Specifiers_{opt}$, $FunName$, $Channels\ldots_{opt}$)" with an initial empty position reserved for custom compiler extensions.

As an attempt to more closely related to the original language proposal (Appendix G), we also split the "`sl_createsync`" construct into two syntax elements "`sl_create`" and "`sl_sync`," bound into a single syntax rule as explained later in Appendix I.5.8.1. Between these two, a new form "`sl_seta`" symmetric to "`sl_geta`" and "`sl_setp`" presented above can provide source values to the channel endpoints. Otherwise, all the positional parameters to "`sl_createsync`" were retained in the definition of "`sl_create`."

Of course, the syntactic substitution of the bulk creation operations for the "fused" interface, or as a C loop for the sequential version, must occur at the point "`sl_sync`" is expressed, not earlier. This is because the source values for channel endpoints may not fully determined prior to this point. This is not further discussed here, as there is no additional technical difficulty involved; for more details cf. the source code of the produced tools, or fig. 6.1 for a summary.

H.9 Support for function calls

Another layer of implementation efforts were dedicated to supporting regular function and procedure calls. These efforts occurred beyond the mandatory provision of private stacks to threads, discussed separately in chapter 9.

The first aspect was to determine how to compile separately a regular C function that can be called from different thread programs. The problem that needed to be addressed is that the register window layout is specific to each thread, and determined by the register configuration word (section 4.3.3.2), whereas the C function code must refer to the same register names in all use contexts. To allow a single code representation for a C function that could be invoked from two threads with different register layout, we had to:

1. constrain the layout of the visible window (section 4.3.3.4) so that the register names in the local synchronizer region would be the same across all threads; this is achieved by mapping the private register space always at the start of the ISA window, i.e. the logical names "$l0...$lN" must map to the machine names "$0...$$N$," instead of "$\$\langle G + S\rangle$...$\$\langle G + S + N\rangle$" as with e.g. the original G-D-S-L layout.
2. apply different post-processing stages to the assembly code of regular C functions than those applied to thread programs, to account for different patterns of register uses; this was achieved by sorting generated assembly codes prior to post-processing based on the presence of the ".registers" directive.

Once function calls can be generated, a situation can occur where a function must save callee-save registers [JR81] before using them for computations. For this purpose the code generator would issue *spills* (stores to stack) in the function code prior to any other use of these registers. However if the function is called from a thread program, and the callee-save registers are not assigned a value in the thread program prior to the function call, it is possible that some of the registers are initially in the "empty" state (section 4.6.1.2), which would incur a deadlock when the spill is reached during execution.

To avoid this problem, we initially modified the post-processor to issue instructions to force all callee-save registers to a non-empty state at the start of a thread program. However, this resulted in a mandatory overhead to fill all the callee-save registers (26 instructions with both the Alpha and SPARC substrate ISAs), even when the registers may end up not being used at all (in most cases it is not possible to determine whether the registers are used or not, since the called function is usually compiled separately). Later on, we negotiated a change to the machine interface semantics instead, to mandate that all local registers are guaranteed upon thread start-up to be readable. This allowed us to avoid the overhead in software.

On a related note, saving and restoring callee-save registers is not necessary in the top-level thread function, since its private register window is dedicated; the post-processor was thus enhanced to remove these when they could be detected.

Finally, when targeting code for the SPARC substrate ISA, we had to address the conceptual conflict between SPARC's sliding register windows [MAG+88] and our machine interface, introduced in section 4.5.4. The problem is that the SPARC machine interface guarantees that the `save` and `restore` instructions will always obtain a "fresh" set of local registers for a function, possibly at the expense of an overflow trap if the window pointer reaches an existing frame around in the register file. Since the implementations we had access to do not implement SPARC's sliding windows and overflow traps, the `save` and `restore` instructions cannot be used at all in thread programs and all the function codes they call indirectly. Instead, we must translate all occurrences of "`save`" and "`restore`" in the assembly source using the substitutions in listings H.25 and H.26. As an optimization, we also automatically skip the saving and restoring instructions that target registers otherwise not used in the current function code.

```
1     save A, B, C
2  →
3     add A, B, %g1
4     std %i0, [%sp - 0]
5     std %i2, [%sp - 8]
6     std %i4, [%sp - 16]
7     std %fp, [%sp - 24]
8     mov %o0, %i0
9     mov %o1, %i1
10    mov %o2, %i2
11    mov %o3, %i3
12    mov %o4, %i4
13    mov %o5, %i5
14    mov %sp, %fp
15    mov %o7, %i7
16    mov %g1, C
```

Listing H.25: Substitution for SPARC's `save`.

```
1     restore A, B, C
2  →
3     add A, B, %g1
4     mov %i7, %o7
5     mov %fp, %sp
6     ldd [%sp - 24], %fp
7     mov %i5, %o5
8     mov %i4, %o4
9     ldd [%sp - 16], %i4
10    mov %i3, %o3
11    mov %i2, %o2
12    ldd [%sp - 8], %i2
13    mov %i1, %o1
14    mov %i0, %o0
15    ldd [%sp - 0], %i0
16    mov %g1, C
```

Listing H.26: Substitution for SPARC's `restore`.

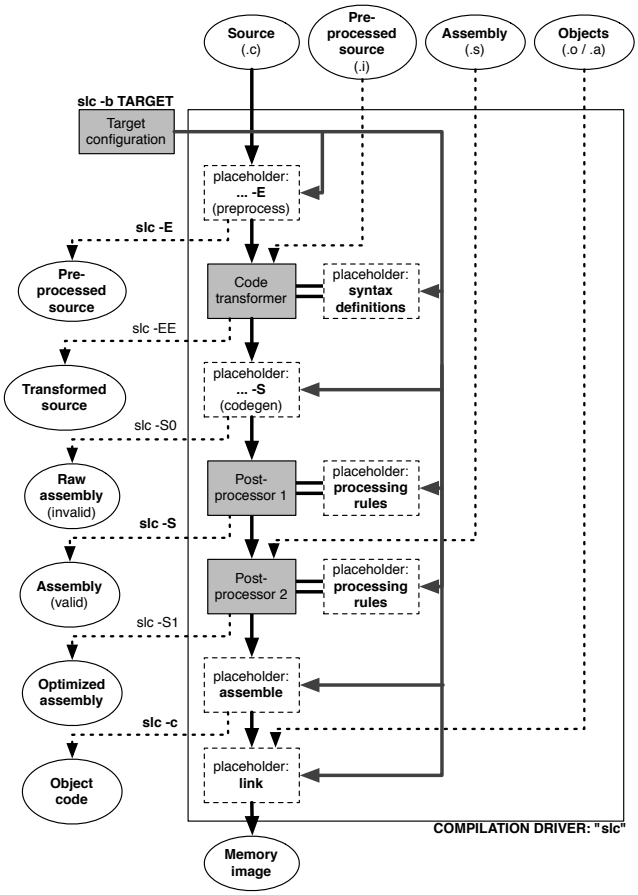


Figure H.3: The "`slc`" compilation pipeline and driver.

H.10 Resulting compilation chain

The resulting tool infrastructure is illustrated in fig. H.3.

Our translation pipeline contains the underlying C pre-processor, our code transformer in charge of performing context-free syntax substitutions on the C source, the underlying C code generator, two stages of post-processing on the assembly code, and the assembler and linker programs. While the previous sections describe only one post-processing stage implementing all transformations on the assembly source, we actually split up this into two stages, the latter containing all transforms that are useful to apply to hand-written assembly. These include the addition of the "swch" annotation, and the removal of family configuration instruction that use the default values in hardware.

This pipeline is further *parameterized* by the target architecture, using the driver command-line parameter "-b": the choice of the underlying C compiler and pre-processor, which syntax substitution and post-processing rules should be applied, and which assembler and linker programs to use can be configured. This way we can use the same tool interface to compile towards various implementations of the new architecture, e.g. to different ISAs, and a

Target alias ("-b")	Description
`mta`	New architecture with the Alpha ISA, MGSim, providing "detached" creation.
`mts`	New architecture with the SPARC ISA, UTLEON3, providing "fused" creation.
`mtsn`	New architecture with the SPARC ISA, MGSim, providing "detached" creation.
`seqc`	Sequential execution on the host platform.
`hrt`	Execution with a software concurrency framework [Mat10], providing "detached" creation.
`ptl, hls`	Execution with two other frameworks [vTJLP09, UvTJ11]; these provide "detached" creation but with "hanging" synchronizer mappings, so they are treated as a "fused" creation interface for code generation.

Table H.3: Supported compilation targets at the time of publication of this book.

legacy "host" platform in pure sequential mode, where the post-processing stages are entirely disabled. A summary of the target configurations we ended up supporting is described in table H.3.

We also took care of recognizing most of the GNU CC command-line interface and external semantics. This allowed us to use the new "`slc`" command as a drop-in replacement, *without changes to existing build systems*, when we ported third-party software programs and libraries. In particular our driver recognizes the common flags "-E" to effect pre-processing only, "-S" to stop after code generation (but ensuring the assembly source is valid for the target architecture), and "-c" to stop after producing object code. The extra flags "-EE," "-S0" and "-S1" are provided for completeness and ease of troubleshooting the compilation chain itself.

Appendix I

SL Language specification

Abstract

This appendix describes the SL language introduced in chapter 6 in the style of [II99, II11b].

Contents

I.1 Prologue

This appendix contains references to terms, definitions and concepts defined in [II99, II11b], which should therefore be considered bound to this specification.

I.2 Terms and definitions

The following terms are defined in section 3 of [II99, II11b] and are reused in this specification: "argument," "constraint," "implementation," "implementation limit," "object," "parameter," "undefined behavior," "unspecified behavior," "value."

The following terms are syntactic / semantic constructs defined in [II99, II11b], section 6, and are reused in this specification: *block item*, *compound statement*, *constant expression*, conversion, *declaration specifier*, *declaration*, *definition*, *expression*, *function designator*, *identifier*, linkage, name space, processing environment, scope, side effect, *statement*, *storage class specifier*, storage duration, translation environment, type, type compatibility, *type specifier*, universal characters, visibility.

As a clarification to [II99, II11b], we further detail the semantics of C with regards to objects, storage duration (lifetime), addressability and mutability in Appendix F.

We did not consider the concurrency features in [II11b] as our work predates this specification. Instead, with [II99] we define the following extra terms:

logical thread a sequential unit of work defined at run-time by a thread program entry point and a channel endpoint interface definition;

family the set of logical threads defined by the execution of a single create construct; this corresponds to the definition of "family" in section 7.2.2.2;

logical thread index integer value that uniquely identifies a thread within a family; the set of logical thread indexes is defined when execution reaches a create construct.

thread program a program suitable for execution by a logical thread;

thread function synonymous to "thread program".

thread parameter an incoming "global" channel endpoint in a thread program, or a pair of incoming/outgoing "shared" channel endpoints.

thread argument an outgoing "global" channel endpoint, or a pair of outgoing/incoming "shared" channel endpoints, in a creating thread respective to the created family.

parameter or argument kind the kind of channel that the parameter or argument endpoint is connected to, either "shared" (daisy-chained between logical threads) or "global" (from creating thread to all created threads).

thread function prototype the specification of the number, kind and type of thread parameters in a thread function declaration or definition.

sibling endpoints for an incoming channel endpoint, the outgoing endpoint at the other side of the channel; for an outgoing channel endpoint, the incoming endpoints at the other side of the channel.

I.3 Conformance

1 Section 4 from [II99, II11b] applies.

Side note I.1: About the compatibility of our implementation.

Our implementation of SL is a *conforming freestanding implementation* as per [II99, 4§6]. Later in our work we added library features to increase its compatibility as a conforming freestanding implementation as per [II11b, 4§6] and a *conforming hosted implementation* as per [II99, II11b], but this support is yet incomplete. We discuss this further in section 6.4.3.

I.4 Environment

1 SL's translation and processing environments are defined like C's environment.
2 Additionally, M4 is run as an additional pre-processor after C pre-processing and before translation.
3 In the rest of this section, the corresponding clauses from [II99, II11b] apply, unless explicitly modified or extended.

I.4.1 Program execution

1 [II99, 5.1.2.3§1] and [II11b, 5.1.2.3§1] apply.
2 [II99, 5.1.2.3§2] and [II11b, 5.1.2.3§2] apply with "at certain specified points in the execution sequence called sequence points [...]" changed to "at certain specified points in the execution sequence *of a thread program*, called sequence points [...]" That is to say, we restrict this clause so that the visibility of side effects is restricted to the rest of the execution of the same thread program, not concurrent activities.
3 Also, creating or waiting for termination of a family, reading from or writing to inter-thread channel endpoints, or calling a function that does any of those operations are all side-effects.
4 [II11b, 5.1.2.3§3] applies, insofar as the proposed relations apply to evaluations performed by *logical* threads as per our definition in I.2.
5 [II99, 5.1.2.3§3] and [II11b, 5.1.2.3§4] apply.
6 We leave unspecified whether the concept of "signal" exists in SL, and thus whether and how [II99, 5.1.2.3§4] and [II11b, 5.1.2.3§5] apply.
7 [II99, 5.1.2.3§5–7] and [II11b, 5.1.2.3§6–8] apply.

I.4.1.1 Multi-threaded executions and data races

1 [II11b, 5.1.2.4§1–4] apply.
2 [II11b, 5.1.2.4§5–10] do not apply directly: we discuss support for C's "atomic objects" and their synchronizing operations further in chapter 7.
3 [II11b, 5.1.2.4§11–28] apply, insofar as C's atomic objects are not supported directly, and the `kill_dependency` macro is not supported in SL.
4 The abstract machine for SL does not use *memory*, and thus C's "*objects*" as per [II99, II11b] and Appendix F, for *synchronization* between threads. Instead, synchronization between threads, and thus the "*inter-thread happens before*" relation between actions described in [II11b, 5.1.2.4§16], is negotiated by programmable devices (dataflow channel endpoints) that are physically independent from memory. This divergence from [II11b] entails that operations that write to an object are not necessarily visible to subsequent read operations, even if the write and read operations are otherwise synchronized. We discuss this further in chapter 7.

I.4.2 Translation limits

1 C's translation limits apply.
2 Also, the implementation shall be able to translate and execute at least one program that contains at least one instance of every one of the following limits:

- 127 "global" channel endpoint definition in a thread program declaration or definition, or family creation;
- 1 "shared" channel endpoint pair definition in a thread program declaration or definition, or family creation;
- 0 levels of syntactic nesting of "`sl_create`" and "`sl_sync`" constructs, that is programs where the program text between the two constructs does not contain further occurrences of the constructs.

I.5 Language

I.5.1 Notation

1 We will reuse the syntax notation from [II99, II11b].
2 In addition, we will use the symbol "¬" to denote continuation of a syntax pattern on the following line.

I.5.2 Concepts

I.5.2.1 Scope of identifiers

1 *Identifiers* can denote what they denote in C ([II99, 6.2.1], [II11b, 6.2.1]);
2 Also, identifiers can denote thread parameters and arguments.
3 Identifiers have the same *visibility* and *scope* semantics in SL as they have in C.

I.5.2.2 Linkage of identifiers

1 *Linkage* is defined in SL as in C ([II99, 6.2.2], [II11b, 6.2.2]).
2 Identifiers that denote channel endpoints have no linkage.

I.5.2.3 Name space of identifiers

1 *Name spaces* are defined in SL as in C for all identifiers except those that denote channel endpoints ([II99, 6.2.3], [II11b, 6.2.3]).
2 Identifiers for channel endpoints have a distinct name space from C label names, tags, members and ordinary identifiers.
3 Also, identifiers that denote thread parameters have a different name space than identifiers that denote thread arguments.

For example in the following fragment:

```
1 sl_def(foo, void, sl_glparm(int, x)) {
2     float x;
3     sl_create(,,,,,, foo, sl_glarg(int, x));
4     sl_sync();
5     // <-- here
```

```
6 }
7 sl_enddef
```

at the location specified by "here," both the variable "x," the thread parameter "x" and the thread argument "x" can be used (the latter two resp. with "`sl_getp`" and "`sl_seta`", cf. hereafter).

I.5.2.4 Storage duration of objects

1 *Storage duration* is defined in SL as for C objects ([II99, 6.2.4], [II11b, 6.2.4]), insofar as [II11b, 6.2.4§4,6.7.1]'s "`_Thread_local`" storage-class specifier is not supported in SL.

(note that since channel endpoints are not objects, they do not have a storage duration)

I.5.2.5 Types

1 *Types* are defined in SL as in C ([II99, 6.2.5], [II11b, 6.2.5]); however the "`_Atomic`" qualifier from [II11b, 6.2.5§27] is not supported in SL as discussed in chapter 7.
2 Also, SL adds a new type category: *thread function types*, which are separate from C's function types. A thread function type is characterized by its attributes and the number, *kind* ("shared" or "global") and type of its thread parameters, collectively called its prototype.
3 A thread function type shall not be incomplete.
4 Thread function types can be used for type derivation like function types.
5 Any of C's *scalar types* ([II99, 6.2.5§21], [II11b, 6.2.5§21],) can be used to define thread parameters and arguments[1].

We furthermore intendedly leave *unspecified* whether channels can be defined using any of C's other types (arrays, aggregates, functions), pending further research in that direction.

I.5.2.6 Representation of types

1 The *representation of types* is defined in SL as in C ([II99, 6.2.6], [II11b, 6.2.6]).

(since channel endpoints themselves are not objects, they do not have a representation, although the values read from or written to the endpoints do have a representation)

I.5.2.7 Compatible types and composite types

1 *Compatible and composite types* are defined in SL as in C ([II99, 6.2.7], [II11b, 6.2.7]).
2 Additionally, compatibility and composition is extended to *thread function types*. However:

- thread function types and C's function types are not mutually compatible;
- two thread function types are not compatible if their attributes differ, of if their argument kind and type differ in any manner when compared one to one in their order of appearance in the function's prototype.

[1] In C, a pointer-to-array type is a scalar type, as well as pointer-to-function.

I.5.2.8 Alignment of objects

1 [II11b, 6.2.8§1] applies; however the `_Alignas` keyword is not supported.
2 [II11b, 6.2.8§2–4] apply, insofar as `_Alignof` and `max_align_t` are not supported.
3 [II11b, 6.2.8§5–7] apply.

I.5.3 Conversions

1 *Conversions* are defined in SL as in C ([II99, 6.3], [II11b, 6.3]).
2 Additionally, automatic conversions are extended to *thread function types* as follows: unlike C function designators, a thread function designator with type "thread function" is never promoted automatically to an expression that has type "pointer to thread function."

I.5.4 Lexical elements

1 *Lexical elements* are defined in SL as in C ([II99, 6.4], [II11b, 6.4]), insofar as the universal character names and string literals from [II11b, 6.4.3,6.4.5] are not supported.
2 Additionally:

- all identifiers beginning with "`__sl_`" and "`_sl_`" are reserved and cannot be used in programs;
- the following pre-processor macros have pervasive semantics and must be considered as keywords (i.e. they must not be redefined, and they can be assumed to have the same semantics everywhere they appear): `sl_def`, `sl_enddef`, `sl_create`, `sl_sync`, `sl_shparm`, `sl_glparm`, `sl_shfparm`, `sl_glfparm`, `sl_glarg`, `sl_glarg`, `sl_shfarg`, `sl_glfarg`, `sl_seta`, `sl_geta`, `sl_setp`, `sl_getp`, `sl_index`, `sl_decl_fptr`, `sl_lbr`, `sl_rbr`, `sl_typedef_fptr`.

3 Any keyword in C^{++} [II11a] not listed above is reserved in SL and cannot be used in programs[2].

I.5.5 Expressions

1 *Expressions* are defined in SL as in C ([II99, 6.5], [II11b, 6.5]), insofar as atomic objects (discussed above), [II11b, 6.5.1.1]'s generic selection with "`_Generic`" and [II11b, 6.5.3]'s alignment operator "`_Alignof`" are not supported in SL.
2 Additionally, SL introduces *thread argument and parameter access expressions.*

I.5.5.1 Thread argument and parameter access

Syntax

primary-expression:
...
`sl_geta (` *identifier* `)`
`sl_getp (` *identifier* `)`

[2] We reserve C^{++} keywords to reserve ourselves the ability to translate SL code to a C^{++}-based substrate language without running the risk that valid uses of identifiers in the source SL code become invalid uses of keywords in C^{++}.

Constraints

1 The identifier used with `sl_geta` must be a visible thread argument name (cf. I.5.8.1).
2 The `sl_geta` expression shall not appear within the create block item list of the create construct where the thread argument name is defined.
3 The identifier used with `sl_getp` must be a thread parameter name in the enclosing thread function.

Semantics

4 Thread argument and parameter access expressions are converted during evaluation to the value read from the corresponding channel endpoint; they correspond to the r and $\bar{q}$ operations from section 7.2.
5 The type of a thread argument or parameter access expression is the declared type of the channel.
6 Each use of `sl_getp` generates a side effect (as per [II99, 5.1.2.3§2] and [II11b, 5.1.2.3§2]).
7 Execution passes the first sequence point following an occurrence of `sl_getp` no earlier than execution passes the `sl_setp` or `sl_seta` statement in the thread where the sibling channel endpoint is defined.
8 If execution reaches an expression using `sl_getp` after it has passed a `sl_setp` statement using the same thread parameter identifier, the behavior of the program becomes undefined.

(note that the *identifiers* themselves that denote thread arguments and parameters are not valid expressions)

I.5.6 Constant expressions

1 *Constant expressions* are defined in SL as in C ([II99, 6.6], [II11b, 6.6]).

I.5.7 Declarations

1 Any C declarations containing the storage class specifier "`typedef`," built upon any of C's types (i.e. not thread function types), any C declarations that declare or define an object or function, and any C declarations that declare a structure, enum or union type are valid in SL and have the same semantics as in C, insofar as the following features from [II11b] are not supported (cf. chapter 7):

- [II11b, 6.7.2.4,6.7.3]'s "`_Atomic`" type specifier and qualifier;
- [II11b, 6.2.4§3,6.7.1]'s "`_Thread_local`" storage class specifier;
- [II11b, 6.7.4]'s "`_Noreturn`" function specifier;
- [II11b, 6.7.5]'s "`_Alignas`" alignment specifier.

2 It is left unspecified here whether an implementation of SL must support [II99, II11b]'s variable length arrays or variably modified types.

(in [II99] this support was mandatory, but it is now an optional feature according to [II11b]. Our implementation makes a "best effort" in this direction but support for this feature was not thoroughly tested.)

3 Additionally, SL adds new constructs for *logical thread index declarations*, *thread function declarations*, *declarations of pointers to thread functions*, and *type name definitions for pointer types to thread functions*.

We intendedly leave *unspecified* whether any other of C's declarations exists in SL.

I.5.7.1 Logical thread index declarations

Syntax

thread-index-declaration:
`sl_index (` *identifier* `) ;`
declaration:
...
thread-index-declaration

Constraints

1 A thread index declaration shall only appear in a thread function body.

Semantics

2 A thread index declaration declare a constant object of unspecified integer type in the current scope with the given name, whose value during execution is the logical thread index of the current thread.

I.5.7.2 Thread function declarations

Syntax

thread-function-declaration:
`sl_decl (` *identifier* `,` $\textit{thread-specifiers}_{opt}$ ¬
`[` `,` *thread-parameter-list* `]`$_{opt}$ `) ;`
thread-parameter-list:
thread-parameter-declaration
thread-parameter-declaration `,` *thread-parameter-list*
external-declaration:
...
thread-function-declaration

(note that this syntax is not suitable for use in a *struct-declaration-list*)

Semantics

1 A thread function declaration declares a thread function with the specified name and prototype, with external linkage unless the attribute "`sl__static`" is specified (cf. I.5.7.5).

I.5.7.3 Declarations of pointers to thread functions

Syntax

thread-function-pointer-declaration:
`sl_decl_fptr (` *identifier* `,` $\textit{thread-specifiers}_{opt}$ ¬
`[` `,` *thread-parameter-list* `]`$_{opt}$ `) ;`
declaration:
...
thread-function-pointer-declaration

(note that this syntax is not suitable for use in a *struct-declaration-list*)

Semantics

1 A thread function pointer declaration declares a pointer to thread function with the specified identifier and prototype, with external linkage unless the attribute "`sl__static`" is specified (cf. I.5.7.5).

I.5.7.4 Type name definitions for pointer types to thread functions

Syntax

thread-function-pointer-typedef:
`sl_typedef_fptr (` *identifier*
`[ ,` *thread-specifiers*$_{opt}$ ¬
`[ ,` *thread-parameter-list* `]`$_{opt}$ `]`$_{opt}$ `) ; ¬`
declaration:
...
thread-function-pointer-typedef

(note that this syntax is suitable for use within function bodies)

Semantics

1 A thread function pointer typedef define a typedef name that denotes a pointer to thread function type with the specified name and prototype.

For example:

```
1 sl_decl(foo,, sl_glparm(int, x));
2 ...
3 {
4     sl_typedef_fptr(ptype,, sl_glparm(int, y));
5     ptype p = &foo;
6     struct { ptype q; } r;
7     r.q = p;
8 }
```

I.5.7.5 Thread attributes and specifiers

Syntax

1 *thread-specifiers*:
thread-specifier-item
`(` *thread-specifier-list* `)`
thread-specifier-list:
thread-specifier-item
thread-specifier-list `,` *thread-specifier-item*
thread-specifier-item:
thread-specifier

thread-attribute
thread-specifier:
`sl__static`
thread-attribute:
(yet undefined)

Semantics

2 The declaration or definition of a thread function can specify *attributes* and *specifiers*. Attributes are part of the prototype, while specifiers are not.
3 The thread specifier `sl__static` plays the same role as C's storage qualifier `static` on external declarations.

I.5.7.6 Thread parameter list

Syntax

thread-parameter-list:
thread-parameter-declaration
thread-parameter-list , *thread-parameter-declaration*
thread-parameter-declaration:
`sl_glparm` (*declaration-specifiers* , *identifier*)
`sl_glfparm` (*declaration-specifiers* , *identifier*)
`sl_shparm` (*declaration-specifiers* , *identifier*)
`sl_shfparm` (*declaration-specifiers* , *identifier*)

Semantics

1 A thread parameter declaration specifies channel endpoints for the thread program. The forms `sl_glparm` and `sl_glfparm` specify an incoming "global" channel endpoint; the forms `sl_shparm` and `sl_shfparm` specify an incoming/outgoing pair of "shared" channel endpoints.
2 The declaration specifiers part of a thread parameter declaration shall not contain a storage class specifier, nor the type qualifier `volatile`.
3 The type designated by the declaration specifiers (either directly, or indirectly via the use of a typedef name) in `sl_glparm` and `sl_shparm` shall be an integer type ([II99, 6.2.5§17], [II11b, 6.2.5§17]).
Note that this includes pointers; as per section 7.3, care must be taken to organize the consistency of the objects pointed to separately.
4 The type designated by the declaration specifiers in `sl_glfparm` and `sl_shpfarm` shall be either `float` or `double`.

I.5.8 Statements and blocks

1 *Statements and compound statements* are defined in SL as in C ([II99, 6.8], [II11b, 6.8]).
2 Also, SL adds *create constructs* and *outgoing communication statements* to C blocks.

I.5.8.1 Family creation

Syntax

1 *create-construct*:
`sl_create` `(` , *create-parameters* , *create-specifiers*$_{opt}$, ¬
assignment-expression ¬
[, *thread-argument-list*]$_{opt}$ `)` ; ¬
create-block-item-list$_{opt}$ ¬
`sl_sync` `(` `)` ;
create-parameters:
assignment-expression$_{opt}$, *range-parameters*
range-parameters:
assignment-expression$_{opt}$, *assignment-expression*$_{opt}$, ¬
assignment-expression$_{opt}$, *assignment-expression*$_{opt}$
create-specifiers:
create-specifier
`(` *create-specifier-list* `)`
create-specifier-list:
create-specifier
create-specifier-list , *create-specifier*
create-specifier:
thread-attribute
thread-argument-list:
thread-argument-definition
thread-argument-list , *thread-argument-definition*
thread-argument-definition:
`sl_glarg` `(` *declaration-specifiers* , *identifier*$_{opt}$ ¬
[, *assignment-expression*]$_{opt}$ `)`
`sl_glfarg` `(` *declaration-specifiers* , *identifier*$_{opt}$ ¬
[, *assignment-expression*]$_{opt}$ `)`
`sl_sharg` `(` *declaration-specifiers* , *identifier*$_{opt}$ ¬
[, *assignment-expression*]$_{opt}$ `)`
`sl_shfarg` `(` *declaration-specifiers* , *identifier*$_{opt}$ ¬
[, *assignment-expression*]$_{opt}$ `)`
create-block-item-list:
create-block-item
create-block-item-list *create-block-item*
create-block-item:
statement
create-construct
block-item:
...
create-construct

2 The syntax form is described as follows: the word "`sl_create`" followed by an opening parenthesis, followed by a comma, followed by five optional assignment expressions separated by (mandatory) commas, followed by an optional create specifier list, followed by a comma,

Side note I.2: About the syntax of "`sl_create`" and "`sl_sync`".

The proposed syntax fuses the "`sl_create`" construct with the "`sl_sync`" construct in a syntactic unit. This formalizes the fundamental distinction between SL and the original language proposal in Appendix G.
The proposed construct also differs from C statements in that it can only appear in an enclosing block (e.g. a function body or compound statement), not as a lone statements where these are allowed (e.g. as one branch of a `if` statement). This is because it plays both the role of a declaration and a statement, and declarations are not valid where lone statements are allowed. Also, intuitively, as declaration, the identifiers they define must be reusable beyond the construct in the enclosing block.
Also, the construct defines a new phrase structure, the *create block item list*: this differs from C's *block item list* ([II99, 6.8.2], [II11b, 6.8.2]) in that it does not allow interspersed declarations, unless they are contained in a compound statement.

Side note I.3: About pointers to thread functions.

In C, an expression with type "pointer to function" can be used in a function call expression, and it is automatically converted to designated the function pointed to. In SL, the thread program expression in a create construct must designate the thread function directly. To use a pointer to thread function, the pointer must be explicitly dereferenced with the unary `*` operator.

followed by an assignment expression, followed by an optional list of comma-separated thread argument definitions, followed by a closing parenthesis, followed by a semicolon, followed by an optional create block item list, followed by the word "`sl_sync`," followed by an opening parenthesis, followed by a closing parenthesis, followed by a semicolon.

3 The first five (optional) assignment expressions are called collectively *family configuration expressions* and individually the *place*, *start*, *limit*, *step*, *block* expressions of the construct. The sixth assignment expression is called the *thread program expression.*

4 In the thread argument list, each argument definition contains declaration specifiers, an optional identifier and an optional assignment expression, separated by commas. The optional identifier is called the *argument name* and the optional assignment expression is called the *argument initializer.*

Constraints

5 Each of the family configuration expressions, when specified, shall have an arithmetic type.

6 The thread program expression must have a thread function type which is compatible with the thread argument list in the number, kind and type of channel definitions.

7 Each argument initializer shall have a type such that its value may be assigned to an object with the unqualified version of the type of its corresponding declaration specifier.

8 The argument names shall not be used in any other create construct in the same scope.

9 A `goto` or `switch` statement expressed outside a create construct shall not refer to a label expressed within its create block item list; nor shall a `goto` or `switch` statement expressed within the create block item list refer to a label outside of the create construct.

10 A `break` or `continue` statement shall only appear within the create block item list of a create construct if the entire enclosing loop also appears inside the construct.

11 The `return` statement shall not be expressed in a create block item list.

(we leave implementation-defined whether any other statements than thread argument assignments and thread parameter assignments with a single indentifier as right operand are valid in a create block item list)

Semantics

12 There is a sequence point after each family configuration expression, after the thread program expression, after each argument initializer, before execution passes the "`sl_create`" into the inner create block item list, before execution reaches "`sl_sync`", and before execution passes "`sl_sync`."

13 If any expression used in the create construct has side effects, they will be effected before their respective sequence point is passed during execution. This includes initialization values for thread parameters that are present in the thread argument list.

14 A create construct maps to a c and s actions in the formalism from section 7.2: both are guaranteed to occur at some point after execution passes "`sl_create`" into the inner create block item list, and before execution passes "`sl_sync`."

15 As a property of the family composition contract, the create construct defines a set of executions of the thread program, collectively called a *family*; each execution in the family is guaranteed to start no earlier than when execution in the creating thread passes beyond the "`sl_create`" part and into the create block item list (or the "`sl_sync`" part if the block item list is not expressed); and execution in the creating thread is guaranteed to pass beyond the last sequence point in the construct no earlier than the termination of all executions in the family.

16 It is implementation-defined whether the c action in the abstract machine occurs *before* execution reaches "`sl_sync`." In particular, the side effects in the create block item list may dominate the start of the family in the precedence order $\rightsquigarrow$; in this case, if execution never reaches that point in the creating thread, the family may not execute at all.

17 Each execution is associated with a unique *logical thread index*, which can be observed via a `sl_index` declaration in the designated thread program (cf. I.5.7.1).

18 The set of all logical thread indexes $\mathcal{S}$ used the family is defined by the triplet of integer values (A, B, C), defined by the integer conversion ([II99, 6.3.1.1], [II11b, 6.3.1.1]) of the start, limit, step configuration expressions, as follows:

$$\mathcal{S} = \begin{cases} \{A + nC \| n \in \mathbb{N} \wedge A \leq A + nC \leq B\} & \text{if } C > 0 \\ \{A + nC \| n \in \mathbb{N} \wedge B \leq A + nC \leq A\} & \text{if } C < 0 \end{cases}$$

The behavior of the program is undefined if $\mathcal{S}$ is not finite when execution reaches the create construct (i.e. when $C = 0$), or if any of the values in $\mathcal{S}$ does not fit in one of the base integer types. If $\mathcal{S} = \varnothing$, no execution of the thread program takes place (in particular when $A = B$).

19 $\mathcal{S}$ is ordered with the following total order:

$$\forall x \in \mathcal{S}, \forall y \in \mathcal{S} \quad x \prec y \Leftrightarrow \begin{cases} x < y & \text{if } C > 0 \\ x > y & \text{if } C < 0 \end{cases}$$

20 The logical index value $x \in \mathcal{S}$ such that $\forall y \in \mathcal{S} \backslash \{x\} \quad x \prec y$, if it exists, is called the *first index* of the family.

21 The logical index value $x \in \mathcal{S}$ such that $\forall y \in \mathcal{S} \backslash \{x\} \quad y \prec x$, if it exists, is called the *last index* of the family.

(the first and last indexes may be identical, e.g. when $A = 0, B = 1, C = 1$)

22 For any logical index value $x \in \mathcal{S}$, its *successor value* is the logical index value $y \in \mathcal{S} \backslash \{x\}$ such that $\forall z \in \mathcal{S} \backslash \{x, y\}, z \prec x \vee y \prec z$. The last index has no successor.

23 The order $\prec$ described here corresponds to the order $\prec$ from section 7.2.2.2: the successor index value of a thread is the index of the successor thread in the family.

Side note I.4: Defining the index sequence in the abstract semantics.

The phrasing in clause I.5.8.1§18 purposefully avoids defining *how* the index sequence is computed, in order to prevent assumptions about the behavior of the machine in case of overflow. This way, we are able to abstract the index sequence away from the operational semantics of a particular implementation.
However it is compatible with the phrasing in section 4.3.2.3.

24 The executions in the family and the execution of the creating thread are related via channels, with a topology determined by the kind of each channel declaration:

- the outgoing endpoint of a "global" channel in the creating thread is connected to the incoming endpoint of the corresponding "global" channel in each execution of the designated thread program;
- the outgoing endpoint of a "shared" channel in the creating thread is connected to the incoming endpoint of the corresponding "shared" channel pair in the execution associated with the first index of the family;
- the outgoing endpoint of a "shared" channel pair in an execution associated with any index but the last is connected to the incoming endpoint of the corresponding "shared" channel pair in the execution associated with the successor index;
- the outgoing endpoint of a "shared" channel pair in an execution associated with the last logical index is connected to the incoming endpoint of the corresponding "shared" channel in the creating thread.

An illustration of this topology is given in fig. I.1 and fig. I.2; control dependencies are given in dotted lines,"global" channels are indicated in blue and "shared" channels in orange. Although the schedule is unspecified, and the family thread executions can occur either simultaneously, in sequence, or out of index order (e.g. in fig. I.2), the channel topology and the relationship between "`sl_set`" and "`sl_get`" is guaranteed by the language semantics.

25 The run-time value b of the "block" family configuration expression, if it is non-zero, *constrains* the definition of the sequential segments of the family (section 7.2.2.2) by the execution environment, by requiring that no more than b separate sequential segments are defined for the family, per processor. This corresponds to the constrain mechanism introduced in section 4.3.2.2. This implies that no more than $b \times P$ units of fair scheduling as used to run the threads in the family, where P is the actual number of processors involved in the creation at run-time. If $b = 0$, no constraint is expressed.

26 The "place" family configuration expressions does not influence the functional behavior of the program; it is described further in chapter 11.

I.5.8.2 Outgoing communication statements

Syntax

thread-argument-assignment:
`sl_seta (` *identifier* `,` *assignment-expression* `) ;`
thread-parameter-assignment:
`sl_setp (` *identifier* `,` *assignment-expression* `) ;`
statement:
...
thread-argument-assignment
thread-parameter-assignment

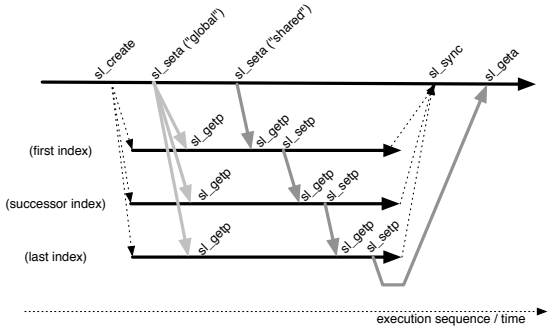


Figure I.1: Channel topology in a parallel thread family.

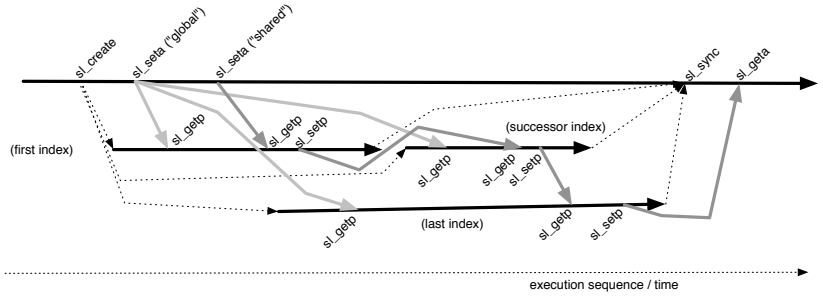


Figure I.2: Channel topology in a thread family with irregular schedule.

Constraints

1 The identifier used with `sl_seta` must be a visible thread argument name (cf. I.5.8.1).
2 The `sl_seta` statement shall not appear outside of the create block item list of the create construct where the corresponding thread argument name is defined.
3 The identifier used with `sl_setp` must be a thread parameter name in the enclosing thread function.
4 The `sl_setp` statement shall not appear outside of a thread function body.
5 The right assignment expression in either `sl_setp` or `sl_seta` must be a suitable expression for use as the right operand of a virtual simple C assignment expression where the left operand would be an lvalue with the channel type ([II99, 6.5.16.1], [II11b, 6.5.16.1]).

Semantics

6 When they are reached during execution, the thread argument and parameter assignment statements cause the output of the value of the specified expression to the channel endpoint designated by the specified identifier; they correspond to the w and $\bar{w}$ operations in section 7.2.
7 There is a sequence point before execution passes a thread argument or parameter assignment statement.
8 If execution reaches a thread argument or parameter assignment statement after it has passed another such statement designating the same channel endpoint, the behavior of the program becomes undefined.

I.5.9 External definitions

1 Any of C's external declarations that declare or define external objects, any of C's external function declarations and definitions, any of C's external declarations with the storage class specifier `typedef`, and any of C's external declarations that declare a structure, enum or union type ([II99, 6.9], [II11b, 6.9]) are valid in SL with the same semantics as in C, with the same restrictions as in clause I.5.7§1 above.

2 Additionally, SL defines a new construct for *thread function definitions*.

I.5.9.1 Thread function definitions

Syntax

thread-function-definition:
`sl_def (` *identifier* [`,` *attributes*$_{opt}$ ¬
[`,` *thread-parameter-list*]$_{opt}$]$_{opt}$ `)` ¬
compound-statement ¬
`sl_enddef`
external-declaration:
...
thread-function-definition

Constraints

1 The identifier in thread functions definitions is in the same name space as C object and function names. Therefore, [II99, 6.9§3] and [II11b, 6.9§3] apply.

2 The constraints in I.5.7.6 apply.

Semantics

3 A thread function definition specifies the name of the thread function being defined, the identifiers of its channel endpoints, and the body of the thread program. The clauses in I.5.7.6 apply.

I.5.10 Preprocessing directives

1 *Preprocessing directives* are defined in SL as in C ([II99, 6.10], [II11b, 6.10]).

2 In particular, an SL implementation may define any of [II11b, 6.10.8.3]'s conditional feature macros (e.g. `__STDC_NO_ATOMICS__`) with the same semantics.

3 Additionally, M4 [KR77] is run to filter the pre-processed text after pre-processing completes and before translation starts, with the M4 quotes changed to "`[[`" and "`]]`" to avoid conflicting with other uses of punctuation valid in C, and all predefined M4 macros renamed with the "`m4_`" prefix to avoid conflicting with existing C identifiers.

I.6 Library

All of C's reserved identifiers are reserved in SL ([II99, 7.1.3], [II11b, 7.1.3]).

Other library services available in SL are described in section 6.4.3.

Appendix J

QuickSort example

To exercise the dynamic mapping scheme described in chapter 10, we experimented with two implementations of the QuickSort algorithm with a cycle-accurate emulator of the target architecture.

J.1 Benchmark program

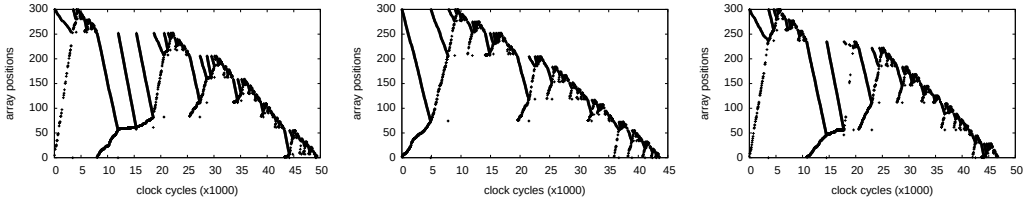
Our two implementations are given in listing J.1 and listing J.2. The first implementation is the "naive" algorithm which spawns two separate families for each branch of the recursion; the second implementation uses a single family of two threads at each recursion step. The procedure implementing each recursion step is augmented to emit the array index of the values being examined: the position of the chosen pivot, and the positions of the successive values being swapped around the pivot.

J.2 Input and baseline

We used three different input arrays, each consisting of 300 pseudo-random integers.

To establish a baseline, we also hand-coded a pure sequential version of the first algorithm which does not use concurrency in any way. We then ran this sequential version on the cycle-accurate machine emulation of the target architecture (MGSim).

Our results are illustrated in fig. J.1. These diagram represent memory activity over time. The horizontal axis represents time; the vertical axis represents cells in the array being sorted. A dot in the diagram indictes a memory read or write operation. The average time to result is between 40 and 50 kcycles.



(a) Data set 1 (b) Data set 2 (c) Data set 3

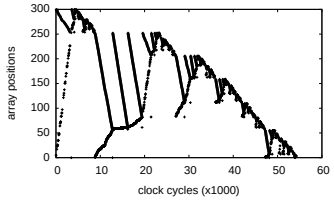
Figure J.1: Baseline: purely sequental algorithm as one thread.

```
1  #include <svp/testoutput.h>
2  #include <stdlib.h>
3  #include <unistd.h>
4  #include <stdio.h>
5  #include <fcntl.h>
6  #define MARK_P1(x) output_uint((x<<2)|1, 0)
7  #define MARK_P2(x) output_uint((x<<2)|2, 0)
8  #define MARK_P3(x) output_uint((x<<2)|3, 0)
9
10 void swap(int *a, int *b) {
11     int t=*a; *a=*b; *b=t;
12 }
13
14 sl_def(sort, sl__static,
15      sl_glparm(int*, arr),
16      sl_glparm(size_t, beg), sl_glparm(size_t, end)) {
17     size_t beg = sl_getp(beg);
18     size_t end = sl_getp(end);
19     if (end > beg + 1) {
20         int *arr = sl_getp(arr);
21
22         /* compute pivot as median of
23            start, end and middle values */
24         int x = arr[beg], y = arr[end-1],
25             z = arr[beg+(end-beg)/2];
26         int piv = ((x < y) && (y < z)) ? y
27                 : (((y < x) && (x < z)) ? x : z);
28         int l = beg + 1, r = end;
29
30         while (l < r) {
31             /* organize values around pivot */
32             MARK_P1(l);
33             if (arr[l] <= piv)
34                 l++;
35             else {
36                 MARK_P2(l); MARK_P3(r-1);
37                 swap(&arr[l], &arr[--r]);
38             }
39         }
40         MARK_P2(l-1); MARK_P3(beg);
41         swap(&arr[--l], &arr[beg]);
42
43         /* recurse */
44         sl_create(,,,,,,,sort, sl_glarg(int*,, arr),
45           sl_glarg(size_t,, beg), sl_glarg(size_t,, l));
46          sl_create(,,,,,,,sort, sl_glarg(int*,, arr),
47           sl_glarg(size_t,, r), sl_glarg(size_t,, end));
48          sl_sync();
49         sl_sync();
50     }
51 }
52 sl_enddef
53
54 int array[100000];
55
56 sl_def(t_main, void) {
57     /* read N values from the file "data",
58        N set via environment variable. */
59     size_t n = atoi(getenv("N"));
60     int fd = open("data", O_RDONLY);
61     for (ssize_t p = 0; p < n; ++p)
62         p += read(fd, array+p, (n-p)*sizeof(int));
63     close(fd);
64
65     /* perform the computation */
66     sl_create(,,,,,,,sort, sl_glarg(int*,,array),
67               sl_glarg(size_t,, 0), sl_glarg(size_t,,n));
68     sl_sync();
69 }
70 sl_enddef
```

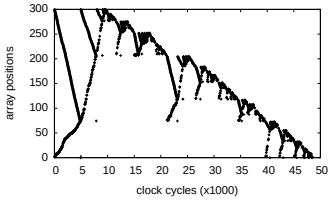
Listing J.1: QuickSort benchmark in SL, classic algorithm.

```
1  #include <svp/testoutput.h>
2  #include <stdlib.h>
3  #include <unistd.h>
4  #include <stdio.h>
5  #include <fcntl.h>
6  #define MARK_P1(x) output_uint((x<<2)|1, 0)
7  #define MARK_P2(x) output_uint((x<<2)|2, 0)
8  #define MARK_P3(x) output_uint((x<<2)|3, 0)
9
10 void swap(int *a, int *b) {
11     int t=*a; *a=*b; *b=t;
12 }
13
14 sl_def(sort2, sl__static, sl_glparm(int*,arr),
15     sl_glparm(size_t, beg1), sl_glparm(size_t, end1),
16     sl_glparm(size_t, beg2), sl_glparm(size_t, end2)) {
17
18     sl_index(i);
19     size_t beg = i ? sl_getp(beg2) : sl_getp(beg1);
20     size_t end = i ? sl_getp(end2) : sl_getp(end1);
21     if (end > beg + 1)
22     {
23         int *arr = sl_getp(arr);
24         /* compute pivot as median of
25            start, end and middle values */
26         int x = arr[beg], y = arr[end-1],
27             z = arr[beg+(end-beg)/2];
28         int piv = ((x < y) && (y < z)) ? y
29                 : (((y < x) && (x < z)) ? x : z);
30         int l = beg + 1, r = end;
31         /* organize values around pivot */
32         while (l < r) {
33             MARK_P1(l);
34             if (arr[l] <= piv)
35                 l++;
36             else {
37                 MARK_P2(l); MARK_P3(r-1);
38                 swap(&arr[l], &arr[--r]);
39             }
40         }
41         MARK_P2(l-1); MARK_P3(beg);
42         swap(&arr[--l], &arr[beg]);
43         /* recurse */
44         sl_create(,,, 2,,,,sort2, sl_glarg(int*,,arr),
45           sl_glarg(size_t,, beg), sl_glarg(size_t,, l),
46           sl_glarg(size_t,, r), sl_glarg(size_t,, end));
47         sl_sync();
48     }
49 }
50 sl_enddef
51
52 int array[100000];
53
54 sl_def(t_main, void) {
55     /* read N values from the file "data",
56        N set via environment variable. */
57     size_t n = atoi(getenv("N"));
58     int fd = open("data", O_RDONLY);
59     for (ssize_t p = 0; p < n; ++p)
60         p += read(fd, array+p, (n-p)*sizeof(int));
61     close(fd);
62
63     /* perform the computation */
64     sl_create(,,,,,,,sort2, sl_glarg(int*,,array),
65             sl_glarg(size_t,, 0), sl_glarg(size_t,, n),
66             sl_glarg(size_t,, 0), sl_glarg(size_t,, 0));
67     sl_sync();
68 }
69 sl_enddef
```

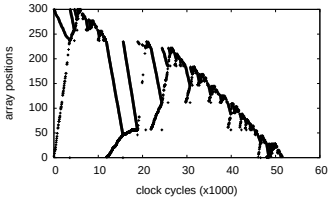
Listing J.2: QuickSort benchmark in SL, families of two threads.



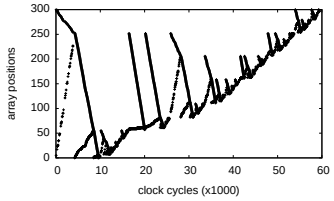
(a) Alg. J.1, Data set 1



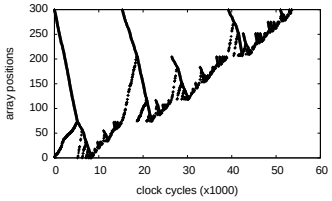
(b) Alg. J.1, Data set 2



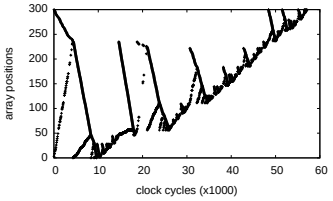
(c) Alg. J.1, Data set 3



(d) Alg. J.2, Data set 1

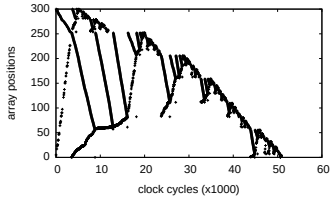


(e) Alg. J.2, Data set 2

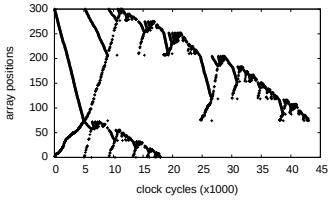


(f) Alg. J.2, Data set 3

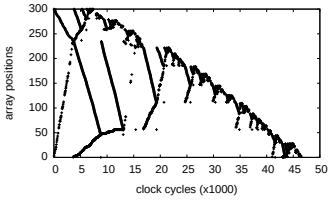
Figure J.2: Execution on 1 core with 1 family context.



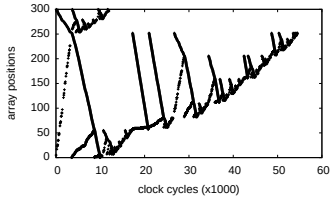
(a) Alg. J.1, Data set 1



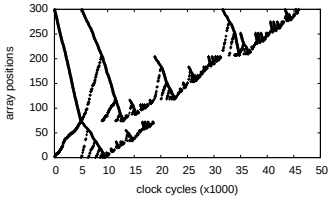
(b) Alg. J.1, Data set 2



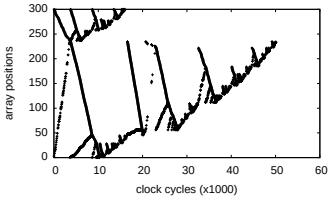
(c) Alg. J.1, Data set 3



(d) Alg. J.2, Data set 1



(e) Alg. J.2, Data set 2



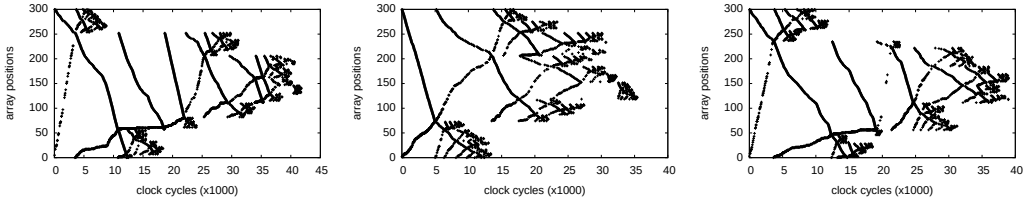
(f) Alg. J.2, Data set 3

Figure J.3: Execution on 1 core with 3 family contexts.

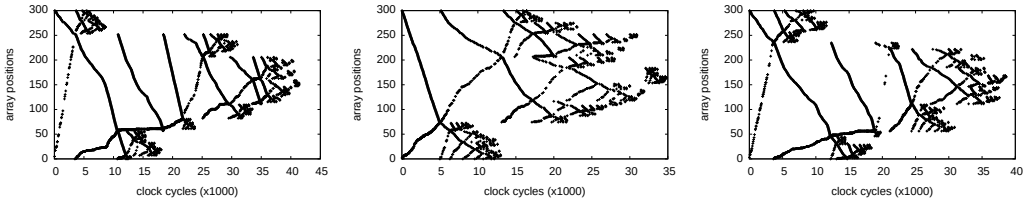
J.3 Multi-threaded behavior

We then executed our concurrent version, with only 1 family context available in the underlying architecture. The results are shown in fig. J.2. The overhead of the conditional on the availability of concurrency resources increases the time to result to between 50 and 60 kcycles.

We then executed the same programs over a system configured to offer only 3 family contexts to programs, on one core. The results are shown in fig. J.3. Despite the overhead,

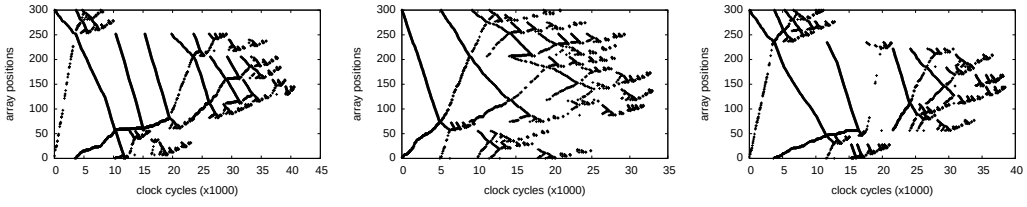


(a) Alg. J.1, Data set 1 (b) Alg. J.1, Data set 2 (c) Alg. J.1, Data set 3



(d) Alg. J.2, Data set 1 (e) Alg. J.2, Data set 2 (f) Alg. J.2, Data set 3

Figure J.4: Execution on 1 core with 31 family contexts.



(a) Data set 1 (b) Data set 2 (c) Data set 3

Figure J.5: Execution on 1 core with 31 family contexts, using Algorithm J.1 and a threshold on concurrency creation.

the overlap of multiple threads in the pipeline reduces bubbles and the total execution time becomes equal or below the overhead.

With 31 family contexts, which is the standard configuration of the target implementation, we obtain the results in fig. J.4 for one core: the time to result is reduced to between 35 and 45 kcycles, i.e. an improvement of 10% to 15% over the baseline.

We also tried to compensate the overhead of testing on resource availability by only using the concurrency construct if the amount of work (size of the sub-array) is larger than 16 items. Our results with this attempt are shown in fig. J.5: although there are less failed allocations overall, the overhead of the additional conditionals (6 extra instructions) is nearly equal to the cost of a failed allocation, resulting in a negligible gain overall.

Appendix K

Mandelbrot set approximation

This appendix provides the source code for the example heterogeneous workload from section 13.3.

```
1  sl_def(mandel,,
2         sl_glfparm(double, four),
3         sl_glfparm(double, xstart), sl_glfparm(double, ystart),
4         sl_glfparm(double, xstep),  sl_glfparm(double, ystep),
5         sl_glparm(uint16_t, xres),  sl_glparm(size_t, icount)) {
6    // compute the complex point from the
7    // logical thread index
8    sl_index(i);
9    uint16_t  xb = i % sl_getp(xres), yb = i / sl_getp(xres);
10   double    cx = sl_getp(xstart) + xb * sl_getp(xstep);
11   double    cy = sl_getp(ystart) + yb * sl_getp(ystep);
12
13   double    zx = cx, zy = cy;
14   size_t    v;
15   for (v = 0; v < sl_getp(icount); ++v) {
16         // iterate z := z^2 + c
17         double q1 = zx * zx, q2 = zy * zy;
18         if ((q1 + q2) >= sl_getp(four))
19             break;
20         double t = q1 - q2 + cx, q3 = zx * zy;
21         zx = t; zy = 2 * q3 + cy;
22   }
23   // pseudo-use of ``v'' to prevent the compiler from erasing the loop as dead code.
24   __asm__ __volatile__("" : : "r"(v));
25 } sl_enddef
```

Listing K.1: Computation kernel executed by each logical thread.

```
1  sl_def(work) {
2    sl_create(,,     /* logical index range: */ 0, NTHREADS_TOTAL, 1,
3              /* max nr. of threads / core: */ THREADS_PER_CORE,,
4              mandel,
5              sl_glfarg(double, , 4.0),
6              sl_glfarg(double, , X_START), sl_glfarg(double, , Y_START),
7              sl_glfarg(double, , X_STEP), sl_glfarg(double, , Y_STEP),
8              sl_glarg(uint16_t, , X_NPOINTS), sl_glarg(size_t, , MAXITER));
9    sl_sync();
10 } sl_enddef
```

Listing K.2: Workload implementation using an even distribution.

```
1  sl_def(mandelouter,,
2         sl_glfparm(double, xstart),
3         sl_glfparm(double, ystart),
4         sl_glfparm(double, xstep),
5         sl_glfparm(double, ystep),
6         sl_glparm(size_t, npoints),
7         sl_glparm(size_t, blocksize),
8         sl_glparm(uint16_t, xres),
9         sl_glparm(size_t, icount))
10 {
11     sl_index(p);
12     size_t n_cores = sl_placement_size(sl_default_placement());
13     sl_create(,
14                 // execute on the local core:
15                 PLACE_LOCAL,
16                 // logical index range:
17                 p, sl_getp(npoints)+p, n_cores,
18                 // number of threads per core:
19                 sl_getp(blocksize), ,
20                 mandel,
21                 sl_glfarg(double, , 4.0),
22                 sl_glfarg(double, , sl_getp(xstart)),
23                 sl_glfarg(double, , sl_getp(ystart)),
24                 sl_glfarg(double, , sl_getp(xstep)),
25                 sl_glfarg(double, , sl_getp(ystep)),
26                 sl_glarg(uint16_t, , sl_getp(xres)),
27                 sl_glarg(size_t, , sl_getp(icount)));
28   sl_sync();
29 }
30 sl_enddef
31
32 sl_def(work)
33 {
34   size_t n_cores = sl_placement_size(sl_default_placement());
35   sl_create(,,
36               // logical index range:
37               0, n_cores, 1,
38               // one thread per core:
39               1,,
40               mandelouter,
41               sl_glfarg(double, , X_START),
42               sl_glfarg(double, , Y_START),
43               sl_glfarg(double, , X_STEP),
44               sl_glfarg(double, , Y_STEP),
45               sl_glarg(size_t, , NTHREADS_TOTAL),
46               sl_glarg(size_t, , THREADS_PER_CORE),
47               sl_glarg(uint16_t, , X_NPOINTS),
48               sl_glarg(size_t, , MAXITER));
49   sl_sync();
50 }
51 sl_enddef
```

Listing K.3: Workload implementation using a round-robin distribution.

Acronyms

ABI	Application Binary Interface
ANSI	American National Standards Institute
ALU	Arithmetic Logic Unit
API	Application Programming Interface
CAS	Compare-and-Swap
CCS	Calculus of Communicating Systems [Mil80]
CMP	Chip Multi-Processor
CSP	Communicating Sequential Processes [Hoa78]
CWP	Current Window Pointer [GAB+88, MAG+88]
DMA	Direct Memory Access
DRAM	Dynamic RAM
ELF	Executable and Linkable Format [Com95]
FIFO	First-In, First-Out
F/OSS	Free and Open Source Software
FP	Floating Point
FPGA	Field-Programmable Gate Array
FPU	Floating-Point Unit
FRF	Floating-point Register File
FU	Functional Unit
GAS	Global Address Space
GC	Garbage Collection
GCC	GNU C Compiler
GCD	Grand Central Dispatch [App]
GFID	Global Family Identifier
GPGPU	General-purpose GPU
GPU	Graphics Processing Unit
ICD	Implicit Communication Domain
CD	Consistency Domain
ILP	Instruction-Level Parallelism
HMT	Hardware Multi-Threading
HIMCYFIO	Here Is My Component, You Figure It Out
HPC	High Performance Computing
IDN	I/O Dynamic Network [BEA+08]
ILP	Instruction-Level Parallelism
IPC	Instructions Per Cycle
IPI	Inter-Processor Interrupt
IRF	Integer Register File
ISA	Instruction Set Architecture
LL/SC	Load-link, Store-conditional
LSB	Least Significant Bit
LSU	Load-Store Unit
MCU	Memory Control Unit
MDN	Memory Dynamic Network [BEA+08]
MFC	Memory Flow Controller [KDH+05]
MIMD	Multiple Instruction, Multiple Data
MMU	Memory Management Unit
MPB	Message-Passing Buffer
MPSoC	Multi-Processor System-on-Chip
MSB	Most Significant Bit
NAS	NASA Advanced Supercomputing
NCU	Network Control Unit
NFS	Network File System
NoC	Network-on-Chip
NUMA	Non-Uniform Memory Access

PC	Program Counter
PPE	Power Processor Element [KDH+05]
QoS	Quality of Service
RAM	Random-Access Memory
RAU	Register Allocation Unit
REST	REpresentational State Transfer [Fie00]
RF	Register File
RISC	Reduced Instruction Set Computer
ROM	Read-Only Memory
RTC	Real-Time Clock
SAC	Single-Assignment C [GS06]
SASOS	Single Address Space Operating System
SCC	Single-Chip Cloud Computer
STN	Static Network [BEA+08]
SM	Streaming Multiprocessor [LNOM08]
SMT	Simultaneous Multi-Threading
SIMD	Single Instruction, Multiple Data
SMP	Symmetric Multi-Processor
SoC	System-on-Chip
SPE	Synergistic Processing Element [KDH+05]
SPEC	Standard Performance Evaluation Corporation
SPU	Synergistic Processing Unit [KDH+05]
SPMD	Single Program, Multiple Data
TBB	Threading Building Blocks [Rei07]
TCU	Thread Creation Unit
TLB	Translation Lookaside Buffer
TLS	Thread-Local Storage
TMU	Thread Management Unit
TPC	Texture/Processor Cluster [LNOM08]
TPL	Thread Parallelism Library [Duf09]
UART	Universal Asynchronous Receiver-Transmitter
UDN	User Dynamic Network [BEA+08]
VLIW	Very Large Instruction Word

Related publications by the author

[1] Thomas Bernard, Clemens Grelck, Michael Hicks, Chris Jesshope, and Raphael Poss. Resource-agnostic programming for many-core Microgrids. In Mario Guarracino, Frédéric Vivien, Jesper Träff, Mario Cannatoro, Marco Danelutto, Anders Hast, Francesca Perla, Andreas Knüpfer, Beniamino Di Martino, and Michael Alexander, editors, *Euro-Par 2010 Parallel Processing Workshops*, volume 6586 of *Lecture Notes in Computer Science*, pages 109–116. Springer Berlin / Heidelberg, 2011. ISBN 978-3-642-21877-4. doi:10.1007/978-3-642-21878-1_14.

[2] Stephan Herhut, Carl Joslin, Sven-Bodo Scholz, Raphael Poss, and Clemens Grelck. Concurrent non-deferred reference counting on the Microgrid: First experiences. In J. Haage and M. Morazán, editors, *Implementation and Application of Functional Languages*, volume 6647 of *Lecture Notes in Computer Science*, pages 185–202. Springer Berlin / Heidelberg, 2011. ISBN 978-3-642-24275-5. doi:10.1007/978-3-642-24276-2_12.

[3] Chris Jesshope, Michael Hicks, Mike Lankamp, Raphael Poss, and Li Zhang. Making multi-cores mainstream – from security to scalability. In Barbara Chapman, Frédéric Desprez, Gerhard R. Joubert, Alain Lichnewsky, Frans Peters, and Thierry Priol, editors, *Parallel Computing: From Multicores and GPU's to Petascale*, volume 19 of *Advances in Parallel Computing*, pages 16–31. IOS Press, 2010. ISBN 978-1-60750-529-7. doi:10.3233/978-1-60750-530-3-16.

[4] Raphael Poss, Clemens Grelck, Stephan Herhut, and Sven-Bodo Scholz. Lazy reference counting for the Microgrid. In *Proc. 16th Workshop on on Interaction between Compilers and Computer Architectures (INTERACT'16)*. IEEE, 2012. (to appear).

[5] Raphael Poss and Chris Jesshope. Towards scalable implicit communication and synchronization. In *The First Workshop on Advances in Message Passing (AMP'10)*. Toronto, Canada, June 2010. Available from: http://www.cs.rochester.edu/u/cding/amp/papers/pos/Towards%20Scalable%20Implicit%20Communication%20and%20Synchronization.pdf.

[6] Raphael Poss, Mike Lankamp, M. Irfan Uddin, Jaroslav Sýkora, and Leoš Kafka. Heterogeneous integration to simplify many-core architecture simulations. In *Proc. 2012 Workshop on Rapid Simulation and Performance Evaluation: Methods and Tools*, RAPIDO '12, pages 17–24. ACM, 2012. ISBN 978-1-4503-1114-4. doi:10.1145/2162131.2162134.

[7] Raphael Poss, Mike Lankamp, Qiang Yang, Jian Fu, Michiel W. van Tol, and Chris Jesshope. Apple-CORE: Microgrids of SVP cores (invited paper). In *Proc. 15th Euromicro Conference on Digital System Design*. IEEE, Cesme, Izmir, Turkey, September 2012. (to appear).

[8] M. Irfan Uddin, Chris R. Jesshope, Michiel W. van Tol, and Raphael Poss. Collecting signatures to model latency tolerance in high-level simulations of microthreaded cores. In *Proceedings of the 2012 Workshop on Rapid Simulation and Performance Evaluation: Methods and Tools*, RAPIDO '12, pages 1–8. ACM, New York, NY, USA, 2012. ISBN 978-1-4503-1114-4. doi:10.1145/2162131.2162132.

Bibliography related to hardware microthreading

[9] R. Abdullaev. A parallel call stack implementation for the microthreaded architecture. Master's thesis, University of Amsterdam, July 2008. Available from: http://dist.svp-home.org/doc/rustam-abdullaev-parallel-stack.pdf.

[10] I. Bell, N. Hasasneh, and C. Jesshope. Supporting microthread scheduling and synchronisation in CMPs. *International Journal of Parallel Programming*, 34:343–381, 2006. ISSN 0885-7458. doi:10.1007/s10766-006-0017-y.

[11] T. Bernard, K. Bousias, B. d. Geus, L. M., L. Zhang, A. Pimentel, P. Knijnenburg, and C. Jesshope. A microthreaded architecture and its compiler. In M. Arenez, R. Doallo, B. B. Fraguela, and J. Tourino, editors, *Proc. 12th Workshop on Compilers for Parallel Computers (CPC'06)*, pages 326–340. University of A Coruña, A Coruña, Spain, January 2006. ISBN 54-609-8459-1.

[12] T. Bernard, K. Bousias, L. Guang, C. R. Jesshope, M. Lankamp, M. W. van Tol, and L. Zhang. A general model of concurrency and its implementation as many-core dynamic RISC processors. In W. Najjar and H. Blume, editors, *Proc. Intl. Conf. on Embedded Computer Systems: Architecture, Modeling and Simulation (IC-SAMOS 2008)*, pages 1–9. IEEE, Samos, Greece, July 2008. ISBN 978-1-4244-1985-2.

[13] T. Bernard, C. Jesshope, and P. Knijnenburg. Strategies for compiling muTC to novel chip multiprocessors. In S. Vassiliadis, M. Berekovic, and T. Hämäläinen, editors, *Embedded Computer Systems: Architectures, Modeling, and Simulation*, volume 4599 of *Lecture Notes in Computer Science*, pages 127–138. Springer Berlin / Heidelberg, 2007. doi:10.1007/978-3-540-73625-7_15.

[14] T. A. M. Bernard. *On the Compilation of a Parallel Language targeting the Self-adaptive Virtual Processor*. PhD thesis, University of Amsterdam, 2010. Available from: http://dare.uva.nl/record/370096.

[15] A. Bolychevsky, C. Jesshope, and V. Muchnick. Dynamic scheduling in RISC architectures. *IEE Proceedings - Computers and Digital Techniques*, 143(5):309–317, September 1996. ISSN 1350-2387. doi:10.1049/ip-cdt:19960788.

[16] K. Bousias, L. Guang, C. Jesshope, and M. Lankamp. Implementation and evaluation of a microthread architecture. *Journal of Systems Architecture*, 55(3):149–161, 2008. doi:10.1016/j.sysarc.2008.07.001.

[17] K. Bousias, N. Hasasneh, and C. Jesshope. Instruction level parallelism through microthreading – a scalable approach to chip multiprocessors. *The Computer Journal*, 49(2):211–233, March 2006. Available from: http://comjnl.oxfordjournals.org/content/49/2/211.abstract, doi:10.1093/comjnl/bxh157.

[18] K. Bousias and C. Jesshope. The challenges of massive on-chip concurrency. In T. Srikanthan, J. Xue, and C.-H. Chang, editors, *Advances in Computer Systems Architecture*, volume 3740 of *Lecture Notes in Computer Science*, pages 157–170. Springer Berlin / Heidelberg, 2005. doi:10.1007/11572961_14.

[19] B. de Graaff. Implementing scheme in a massively concurrent architecture. BSc Thesis, University of Amsterdam, Institute for Informatics, June 2011. Available from: http://dist.svp-home.org/doc/ben-de-graaff-sl-scheme-for-microgrid.pdf.

[20] T. D.Vu, L. Zhang, and C. R. Jesshope. The verification of the on-chip COMA cache coherence protocol. In *International Conference on Algebraic Methodology and Software Technology*, pages 413–429, 2008. ISBN 978-3-540-79979-5.

[21] N. Hasasneh, I. Bell, and C. Jesshope. Asynchronous arbiter for micro-threaded chip multiprocessors. *Journal of Systems Architecture*, 53(5-6):253–262, 2007. ISSN 1383-7621. Available from: `http://www.sciencedirect.com/science/article/pii/S1383762106001263`, `doi:10.1016/j.sysarc.2006.10.004`.

[22] N. M. Hasasneh. *Chip Multi-Processors using a Micro-Threaded model*. PhD thesis, University of Hull, October 2006.

[23] C. Jesshope. Implementing an efficient vector instruction set in a chip multi-processor using micro-threaded pipelines. In *Proc. 6th Australasian Computer Systems Architecture Conference (ACSAC'01)*, pages 80–88. IEEE, 2001. ISBN 0-7695-0954-1. `doi:10.1109/ACAC.2001.903363`.

[24] C. Jesshope. Multi-threaded microprocessors – evolution or revolution. In A. Omondi and S. Sedukhin, editors, *Advances in Computer Systems Architecture*, volume 2823 of *Lecture Notes in Computer Science*, pages 21–45. Springer Berlin / Heidelberg, 2003. `doi:10.1007/978-3-540-39864-6_4`.

[25] C. Jesshope. Scalable instruction-level parallelism. In A. Pimentel and S. Vassiliadis, editors, *Computer Systems: Architectures, Modeling, and Simulation*, volume 3133 of *Lecture Notes in Computer Science*, pages 383–392. Springer Berlin / Heidelberg, 2004. `doi:10.1007/978-3-540-27776-7_40`.

[26] C. Jesshope. Microgrids – the exploitation of massive on-chip concurrency. In L. Grandinetti, editor, *Grid Computing The New Frontier of High Performance Computing*, volume 14 of *Advances in Parallel Computing*, pages 203–223. North-Holland, 2005. ISSN 0927-5452. Available from: `http://www.sciencedirect.com/science/article/pii/S0927545205800127`, `doi:10.1016/S0927-5452(05)80012-7`.

[27] C. Jesshope. Microthreading, a model for distributed instruction-level concurrency. *Parallel Processing Letters*, 16(2):209–228, 2006.

[28] C. Jesshope. A model for the design and programming of multi-cores. In L. Grandinetti, editor, *High Performance Computing and Grids in Action*, number 16 in Advances in Parallel Computing, pages 37–55. IOS Press, 2008. ISBN 978-1-58603-839-7. Available from: `http://dare.uva.nl/record/288698`.

[29] C. Jesshope. Operating systems in silicon and the dynamic management of resources in many-core chips. *Parallel Processing Letters*, 18(2):257–274, 2008.

[30] C. Jesshope, M. Lankamp, and L. Zhang. Evaluating CMPs and their memory architecture. In *Architecture of Computing Systems – ARCS 2009*, volume 5455/2009 of *Lecture Notes in Computer Science*, pages 246–257. Springer Berlin / Heidelberg, 2009. ISBN 978-3-642-00453-7. ISSN 0302-9743 (Print) 1611-3349 (Online). `doi:10.1007/978-3-642-00454-4_24`.

[31] C. Jesshope, M. Lankamp, and L. Zhang. The Implementation of an SVP Many-core Processor and the Evaluation of its Memory Architecture. *ACM SIGARCH Computer Architecture News*, 37(2):38–45, 2009. ISSN 0163-5964. `doi:10.1145/1577129.1577136`.

[32] C. Jesshope and B. Luo. Micro-threading: a new approach to future RISC. In *Proc. 5th Australasian Computer Architecture Conference (ACAC'00)*, pages 34–41. IEEE, 2000. ISBN 0-7695-0512-0. `doi:10.1109/ACAC.2000.824320`.

[33] C. Jesshope, J.-M. Philippe, and M. van Tol. An architecture and protocol for the management of resources in ubiquitous and heterogeneous systems based on the SVP model of concurrency. In *Embedded Computer Systems: Architectures, Modeling, and Simulation*, pages 218–228, 2008. ISBN 978-3-540-70549-9.

[34] C. R. Jesshope. μTC - an intermediate language for programming chip multiprocessors. In C. Jesshope and C. Egan, editors, *Advances in Computer Systems Architecture*, volume 4186 of *Lecture Notes in Computer Science*, pages 147–160. Springer Berlin / Heidelberg, 2006. ISBN 978-3-540-40056-1. `doi:10.1007/11859802_13`.

[35] M. Lankamp. Developing a Reference Implementation for a Microgrid of Microthreaded Microprocessors. Master's thesis, University of Amsterdam, Amsterdam, the Netherlands, August 2007. Available from: `http://dist.svp-home.org/doc/mike-lankamp-ref-microgrid.pdf`.

[36] M. Lankamp. *Design and Evaluation of a Multithreaded Many-Core Architecture*. PhD thesis, University of Amsterdam, 201x. To appear.

[37] M. Lankamp and T. Bernard. Microthreads reference document (draft). Technical report, University of Amsterdam, November 2008.

[38] B. Luo and C. Jesshope. Performance of a micro-threaded pipeline. In *Proc. 7th Asia-Pacific conference on Computer systems architecture (CRPIT '02)*, pages 83–90. Australian Computer Society, Inc., Darlinghurst, Australia, Australia, 2002. ISBN 0-909925-84-4.

[39] A. Matei. Towards adaptable parallel software - the Hydra runtime for SVP programs. Master's thesis, Vrije Universiteit Amsterdam, Faculty of Science, Department of Mathematics and Computer Science, November 2010. Available from: `http://dist.svp-home.org/doc/andrei-matei-hydra-svp-runtime.pdf`.

[40] M. I. Uddin, M. W. van Tol, and C. R. Jesshope. High level simulation of SVP many-core systems. *Parallel Processing Letters*, 21(4):413–438, December 2011. ISSN 0129-6264. `doi:10.1142/S0129626411000308`.

[41] M. W. van Tol. Exceptions in a microthreaded architecture. Master's thesis, University of Amsterdam, December 2006. Available from: `http://dist.svp-home.org/doc/michiel-van-tol-exceptions.ps`.

[42] M. W. van Tol, C. R. Jesshope, M. Lankamp, and S. Polstra. An implementation of the SANE Virtual Processor using POSIX threads. *J. Syst. Archit.*, 55(3):162–169, 2009. ISSN 1383-7621.

[43] M. W. van Tol and J. Koivisto. Extending and implementing the self-adaptive virtual processor for distributed memory architectures. Technical Report arXiv:1104.3876v1 [cs.DC], University of Amsterdam and VTT, Finland, 2011. Available from: `http://arxiv.org/abs/1104.3876`.

[44] T. D. Vu and C. R. Jesshope. Formalizing SANE virtual processor in thread algebra. In M. Butler, M. Hinchey, and M. Larrondo-Petrie, editors, *Formal Methods and Software Engineering*, volume 4789 of *Lecture Notes in Computer Science*, pages 345–365. Springer Berlin / Heidelberg, 2007. ISBN 978-3-540-76648-3. `doi:10.1007/978-3-540-76650-6_20`.

[45] Q. Yang, C. Jesshope, and J. Fu. A micro threading based concurrency model for parallel computing. In *Proc 2011 IEEE International Symposium on Parallel and Distributed Processing Workshops and Phd Forum (IPDPSW)*, pages 1668–1674, May 2011. ISSN 1530-2075. `doi:10.1109/IPDPS.2011.323`.

[46] L. Zhang and C. R. Jesshope. On-Chip COMA Cache-Coherence Protocol for Microgrids of Microthreaded Cores. In Bouge and et al., editors, *Euro-Par Workshops*, volume 4854 of *LNCS*, pages 38–48. Springer, 2007.

General bibliography

[AAR08] Mohamed F. Ahmed, Reda A. Ammar, and Sanguthevar Rajasekaran. Spenk: adding another level of parallelism on the Cell Broadband Engine. In *Proc. 1st international forum on Next-generation multicore/manycore technologies*, IFMT '08, pages 2:1–2:10. ACM, New York, NY, USA, 2008. ISBN 978-1-60558-407-2. doi:10.1145/1463768.1463771.

[Abd08] Rustam Abdullaev. A parallel call stack implementation for the microthreaded architecture. Master's thesis, University of Amsterdam, July 2008. Available from: http://dist.svp-home.org/doc/rustam-abdullaev-parallel-stack.pdf.

[ACH+08] Eric Allen, David Chase, Joe Hallett, Victor Luchangco, Jan-Willem Maessen, Sukyoung Ryu, Guy L. Steele Jr., and Sam Tobin-Hochstadt. The Fortress language specification, version 1.0. Technical report, Sun Microsystems, March 2008. Available from: http://research.sun.com/projects/plrg/fortress.pdf.

[ADNP88] Arvind, Michael L. Dertouzos, Rishiyur S. Nikhil, and Gregory M. Papadopoulos. Project dataflow: A parallel computing system based on the Monsoon architecture and the Id programming language. Computation Structures Group Memo 285, Laboratory for Computer Science, Massachusetts Institute of Technology, March 1988. Available from: http://csg.csail.mit.edu/pubs/memos/Memo-285/Memo-285.pdf.

[Adv] Advanced Micro Devices, Inc. AMD Fusion APU era begins. Available from: http://www.amd.com/us/press-releases/Pages/amd-fusion-apu-era-2011jan04.aspx [cited March 2012].

[Agh85] Gul Abdulnabi Agha. ACTORS: A model of concurrent computation in distributed systems. AITR 844, Massachusetts Institute of Technology, 1985. Available from: http://dspace.mit.edu/handle/1721.1/6952.

[AHKB00] Vikas Agarwal, M. S. Hrishikesh, Stephen W. Keckler, and Doug Burger. Clock rate versus IPC: the end of the road for conventional microarchitectures. *SIGARCH Comput. Archit. News*, 28:248–259, May 2000. ISSN 0163-5964. doi:10.1145/342001.339691.

[ALL89] T.E. Anderson, E.D. Lazowska, and H.M. Levy. The performance implications of thread management alternatives for shared-memory multiprocessors. *IEEE Trans. Comput.*, 38(12): 1631–1644, dec 1989. ISSN 0018-9340. doi:10.1109/12.40843.

[Amd67] Gene M. Amdahl. Validity of the single processor approach to achieving large scale computing capabilities. In *Proc. of the April 18-20, 1967, Spring Joint Computer Conference*, AFIPS '67 (Spring), pages 483–485. ACM, New York, NY, USA, 1967. doi:10.1145/1465482.1465560.

[And] Erik Andersen. μClibc, a C library for embedded Linux. Available from: http://uclibc.org/about.html [cited October 2011].

[ANP87] Arvind, Rishiyur Nikhil, and Keshav Pingali. I-structures: Data structures for parallel computing. In Joseph Fasel and Robert Keller, editors, *Graph Reduction*, volume 279 of *Lecture Notes in Computer Science*, pages 336–369. Springer Berlin / Heidelberg, 1987. doi:10.1007/3-540-18420-1_65.

[App] Apple, Inc. Grand Central Dispatch (GCD) reference. Available from: http://developer.apple.com/mac/library/documentation/Performance/Reference/GCD_libdispatch_Ref/Reference/reference.html [cited September 2011].

[ARMa] ARM. ARM compiler toolchain compiler reference, version 5.0: Compiler-specific features: __asm. Available from: http://infocenter.arm.com/help/topic/com.arm.doc.dui0491e/BABFDCGD.html [cited September 2011].

[ARMb] ARM. ARM compiler toolchain compiler reference, version 5.0: Compiler-specific features: Named register variables. Available from: http://infocenter.arm.com/help/topic/com.arm.doc.dui0491e/CJAHJDBG.html [cited September 2011].

[ARM09] ARM. Procedure call standard for the ARM® architecture. Standard ARM-IHI-0042D, October 2009. Available from: http://infocenter.arm.com/help/topic/com.arm.doc.ihi0042d/IHI0042D_aapcs.pdf.

[AVWW96] Joe Armstrong, Robert Virding, Claes Wikström, and Mike Williams. *Concurrent Programming in ERLANG*. Prentice Hall, 2nd edition, 1996. ISBN 0-13-508301-X.

[Bat68] K. E. Batcher. Sorting networks and their applications. In *Proceedings of the April 30–May 2, 1968, spring joint computer conference*, AFIPS '68 (Spring), pages 307–314. ACM, New York, NY, USA, 1968. doi:10.1145/1468075.1468121.

[BBD+09] Andrew Baumann, Paul Barham, Pierre-Evariste Dagand, Tim Harris, Rebecca Isaacs, Simon Peter, Timothy Roscoe, Adrian Schüpbach, and Akhilesh Singhania. The multikernel: a new os architecture for scalable multicore systems. In *Proc. ACM SIGOPS 22nd symposium on Operating systems principles*, SOSP'09, pages 29–44. ACM, New York, NY, USA, 2009. ISBN 978-1-60558-752-3. doi:10.1145/1629575.1629579.

[BBG+06] T. Bernard, K. Bousias, B. de Geus, Lankamp M., L. Zhang, A.D. Pimentel, P.M.W. Knijnenburg, and C.R. Jesshope. A microthreaded architecture and its compiler. In M. Arenez, R. Doallo, B. B. Fraguela, and J. Tourino, editors, *Proc. 12th Workshop on Compilers for Parallel Computers (CPC'06)*, pages 326–340. University of A Coruña, A Coruña, Spain, January 2006. ISBN 54-609-8459-1.

[BBG+08] T. Bernard, K. Bousias, L. Guang, C. R. Jesshope, M. Lankamp, M. W. van Tol, and L. Zhang. A general model of concurrency and its implementation as many-core dynamic RISC processors. In W. Najjar and H. Blume, editors, *Proc. Intl. Conf. on Embedded Computer Systems: Architecture, Modeling and Simulation (IC-SAMOS 2008)*, pages 1–9. IEEE, Samos, Greece, July 2008. ISBN 978-1-4244-1985-2.

[BCS+98] Jay Boisseau, Larry Carter, Allan Snavely, David Callahan, John Feo, Simon Kahan, and Zhijun Wu. CRAY T90 vs. Tera MTA: The old champ faces a new challenger. In *Proc. Cray User's Group Conference*. Cray Inc., 411 First Avenue South, Seattle, WA 9810, USA, June 1998.

[BEA+08] S. Bell, B. Edwards, J. Amann, R. Conlin, K. Joyce, V. Leung, J. MacKay, M. Reif, Liewei Bao, J. Brown, M. Mattina, Chyi-Chang Miao, C. Ramey, D. Wentzlaff, W. Anderson, E. Berger, N. Fairbanks, D. Khan, F. Montenegro, J. Stickney, and J. Zook. TILE64 processor: A 64-core SoC with mesh interconnect. In *IEEE International Solid-State Circuits Conference, 2008 (ISSCC 2008). Digest of Technical Papers.*, pages 88–598. IEEE, February 2008. doi:10.1109/ISSCC.2008.4523070.

[Bem57] R.W. Bemer. How to consider a computer. *Automatic Control Magazine*, pages 66–69, March 1957. Available from: http://www.trailing-edge.com/~bobbemer/TIMESHAR.HTM.

[Ber09] Leslin Berlin. Kicking reality up a notch. *The New York Times*, July 2009. Available from: http://www.nytimes.com/2009/07/12/business/12proto.html.

[Ber10] Thomas A. M. Bernard. *On the Compilation of a Parallel Language targeting the Self-adaptive Virtual Processor*. PhD thesis, University of Amsterdam, 2010. Available from: http://dare.uva.nl/record/370096.

[BFG+06] K. Bernstein, D. J. Frank, A. E. Gattiker, W. Haensch, B. L. Ji, S. R. Nassif, E. J. Nowak, D. J. Pearson, and N. J. Rohrer. High-performance CMOS variability in the 65-nm regime and beyond. *IBM J. Res. Dev.*, 50(4/5):433–449, July 2006. ISSN 0018-8646.

[BFJ+96] Robert D. Blumofe, Matteo Frigo, Christopher F. Joerg, Charles E. Leiserson, and Keith H. Randall. An analysis of dag-consistent distributed shared-memory algorithms. In *SPAA '96: Proceedings of the eighth annual ACM symposium on Parallel algorithms and architectures*, pages 297–308. ACM, New York, NY, USA, 1996. ISBN 0-89791-809-6. doi:10.1145/237502.237574.

[BGH+11] Thomas Bernard, Clemens Grelck, Michael Hicks, Chris Jesshope, and Raphael Poss. Resource-agnostic programming for many-core Microgrids. In Mario Guarracino, Frédéric Vivien, Jesper Träff, Mario Cannatoro, Marco Danelutto, Anders Hast, Francesca Perla, Andreas Knüpfer, Beniamino Di Martino, and Michael Alexander, editors, *Euro-Par 2010 Parallel Processing Workshops*, volume 6586 of *Lecture Notes in Computer Science*, pages 109–116. Springer Berlin / Heidelberg, 2011. ISBN 978-3-642-21877-4. doi:10.1007/978-3-642-21878-1_14.

[BGJL08] K. Bousias, L. Guang, C.R. Jesshope, and M. Lankamp. Implementation and evaluation of a microthread architecture. *Journal of Systems Architecture*, 55(3):149–161, 2008. doi:10.1016/j.sysarc.2008.07.001.

[BHJ06a] Ian Bell, Nabil Hasasneh, and Chris Jesshope. Supporting microthread scheduling and synchronisation in CMPs. *International Journal of Parallel Programming*, 34:343–381, 2006. ISSN 0885-7458. doi:10.1007/s10766-006-0017-y.

[BHJ06b] Kostas Bousias, Nabil Hasasneh, and Chris Jesshope. Instruction level parallelism through microthreading – a scalable approach to chip multiprocessors. *The Computer Journal*, 49(2): 211–233, March 2006. Available from: http://comjnl.oxfordjournals.org/content/49/2/211.abstract, doi:10.1093/comjnl/bxh157.

[BJ05] Kostas Bousias and Chris Jesshope. The challenges of massive on-chip concurrency. In Thambipillai Srikanthan, Jingling Xue, and Chip-Hong Chang, editors, *Advances in Computer Systems Architecture*, volume 3740 of *Lecture Notes in Computer Science*, pages 157–170. Springer Berlin / Heidelberg, 2005. doi:10.1007/11572961_14.

[BJK+95] Robert D. Blumofe, Christopher F. Joerg, Bradley C. Kuszmaul, Charles E. Leiserson, Keith H. Randall, and Yuli Zhou. Cilk: an efficient multithreaded runtime system. *SIGPLAN Not.*, 30 (8):207–216, 1995. ISSN 0362-1340. doi:10.1145/209937.209958.

[BJK07] Thomas Bernard, Chris Jesshope, and Peter Knijnenburg. Strategies for compiling muTC to novel chip multiprocessors. In Stamatis Vassiliadis, Mladen Berekovic, and Timo Hämäläinen, editors, *Embedded Computer Systems: Architectures, Modeling, and Simulation*, volume 4599 of *Lecture Notes in Computer Science*, pages 127–138. Springer Berlin / Heidelberg, 2007. doi:10.1007/978-3-540-73625-7_15.

[BJM96] A. Bolychevsky, C.R. Jesshope, and V.B. Muchnick. Dynamic scheduling in RISC architectures. *IEE Proceedings - Computers and Digital Techniques*, 143(5):309–317, September 1996. ISSN 1350-2387. doi:10.1049/ip-cdt:19960788.

[BL93] Robert D. Blumofe and Charles E. Leiserson. Space-efficient scheduling of multithreaded computations. In *STOC '93: Proceedings of the twenty-fifth annual ACM symposium on Theory of computing*, pages 362–371. ACM, New York, NY, USA, 1993. ISBN 0-89791-591-7. doi:10.1145/167088.167196.

[BM07] Jan A. Bergstra and C. A. Middelburg. Thread algebra for strategic interleaving. *Formal Aspects of Computing*, 19(4):445–474, 2007. doi:10.1007/s00165-007-0024-9.

[BPS+09] Andrew Baumann, Simon Peter, Adrian Schüpbach, Akhilesh Singhania, Timothy Roscoe, Paul Barham, and Rebecca Isaacs. Your computer is already a distributed system. why isn't your OS? In *Proc. 12th conference on Hot topics in operating systems (HotOS'09)*. USENIX Association, Berkeley, CA, USA, 2009.

[BS06] Jurgen Bitzer and Philipp J.H. Schroder, editors. *The Economics of Open Source Software Development*. Emerald Group Publishing Limited, October 2006. ISBN 978-0444527691.

[BSL+02] Rajeshwari Banakar, Stefan Steinke, Bo-Sik Lee, M. Balakrishnan, and Peter Marwedel. Scratchpad memory: design alternative for cache on-chip memory in embedded systems. In *Proceedings of the tenth international symposium on Hardware/software codesign*, CODES '02, pages 73–78. ACM, New York, NY, USA, 2002. ISBN 1-58113-542-4. doi:10.1145/774789.774805.

[CA88] D. E. Culler and Arvind. Resource requirements of dataflow programs. *SIGARCH Comput. Archit. News*, 16:141–150, May 1988. ISSN 0163-5964. doi:10.1145/633625.52417.

[CBHLL92] Jeff Chase, Miche Baker-Harvey, Hank Levy, and Ed Lazowska. Opal: A single address space system for 64-bit architectures. *SIGOPS Oper. Syst. Rev.*, 26:9–, April 1992. ISSN 0163-5980. Available from: `http://dl.acm.org/citation.cfm?id=142111.964562`.

[CCS98] Bradford Chamberlain, Sung-Eun Choi, and Lawrence Snyder. A compiler abstraction for machine independent parallel communication generation. In Zhiyuan Li, Pen-Chung Yew, Siddharta Chatterjee, Chua-Huang Huang, P. Sadayappan, and David Sehr, editors, *Languages and Compilers for Parallel Computing*, volume 1366 of *Lecture Notes in Computer Science*, pages 261–276. Springer Berlin / Heidelberg, 1998. ISBN 978-3-540-64472-9. `doi:10.1007/BFb0032698`.

[CCZ07] B.L. Chamberlain, D. Callahan, and H.P. Zima. Parallel programmability and the Chapel language. *The International Journal of High Performance Computing Applications*, 21(3): 291–312, August 2007. `doi:10.1177/1094342007078442`.

[CD97a] M. Cekleov and M. Dubois. Virtual-address caches. part 1: problems and solutions in uniprocessors. *IEEE Micro*, 17(5):64–71, sep/oct 1997. ISSN 0272-1732. `doi:10.1109/40.621215`.

[CD97b] M. Cekleov and M. Dubois. Virtual-address caches. part 2: Multiprocessor issues. *Micro, IEEE*, 17(6):69–74, nov/dec 1997. ISSN 0272-1732. `doi:10.1109/40.641599`.

[CDMC+05] Cristian Coarfa, Yuri Dotsenko, John Mellor-Crummey, François Cantonnet, Tarek El-Ghazawi, Ashrujit Mohanti, Yiyi Yao, and Daniel Chavarría-Miranda. An evaluation of global address space languages: Co-Array Fortran and Unified Parallel C. In *PPoPP '05: Proc. 10th ACM SIGPLAN symposium on Principles and practice of parallel programming*, pages 36–47. ACM, New York, NY, USA, 2005. ISBN 1-59593-080-9. `doi:10.1145/1065944.1065950`.

[CGMN80] T. J.W. Clarke, P. J.S. Gladstone, C. D. MacLean, and A. C. Norman. SKIM—the S, K, I reduction machine. In *Proceedings of the 1980 ACM conference on LISP and functional programming*, LFP'80, pages 128–135. ACM, New York, NY, USA, 1980. `doi:10.1145/800087.802798`.

[CGS+05] Philippe Charles, Christian Grothoff, Vijay Saraswat, Christopher Donawa, Allan Kielstra, Kemal Ebcioglu, Christoph von Praun, and Vivek Sarkar. X10: an object-oriented approach to non-uniform cluster computing. In *OOPSLA '05: Proceedings of the 20th annual ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, pages 519–538. ACM, New York, NY, USA, 2005. ISBN 1-59593-031-0. `doi:10.1145/1094811.1094852`.

[Cha01] Bradford L. Chamberlain. *The design and implementation of a region-based parallel programming language*. PhD thesis, University of Washington, November 2001. Available from: `http://www.cs.washington.edu/homes/brad/cv/pubs/degree/thesis.html`.

[Cha02] Paul Chapman. Life universal computer, November 2002. Available from: `http://www.igblan.free-online.co.uk/igblan/ca/` [cited August 2011].

[CHA06] Kevin Crowston, James Howison, and Hala Annabi. Information systems success in free and open source software development: theory and measures. *Software Process: Improvement and Practice*, 11(2):123–148, 2006. ISSN 1099-1670. `doi:10.1002/spip.259`.

[CLFL94] Jeffrey S. Chase, Henry M. Levy, Michael J. Feeley, and Edward D. Lazowska. Sharing and protection in a single-address-space operating system. *ACM Trans. Comput. Syst.*, 12:271–307, November 1994. ISSN 0734-2071. `doi:10.1145/195792.195795`.

[CMN83] Stuart K. Card, Thomas P. Moran, and Allen Newell. *The psychology of human-computer interaction*. L. Erlbaum Associates, 1983. ISBN 0898592437.

[Com95] Tool Interface Standard (TIS) Committee. *Executable and Linking Format (ELF) Specification, Version 1.2*, May 1995. Available from: `http://refspecs.freestandards.org/elf/elf.pdf`.

[Con08] Open Watcom Contributors. *Open Watcom C/C++ User's Guide, Version 1.8*, 2008. Available from: `http://www.openwatcom.org/ftp/manuals/current/cguide.pdf`.

[Coo04] Matthew Cook. Universality in elementary cellular automata. *Complex Systems*, 15(1):1–40, 2004. Available from: `http://www.complex-systems.com/Archive/hierarchy/abstract.cgi?vol=15&iss=1&art=01`.

[Cor] Microsoft Corporation. Visual Studio 2010: C++ language reference: Inline assembler overview. Available from: http://msdn.microsoft.com/en-us/library/4ks26t93.aspx [cited September 2011].

[Cor09] Intel Corporation. An introduction to the Intel® QuickPath Interconnect. Whitepaper 320412-001US, Intel Corporation, January 2009. Available from: http://www.intel.com/content/www/us/en/io/quickpath-technology/quick-path-interconnect-introduction-paper.html.

[Cra11] Cray Inc, 901 Fifth Avenue, Suite 100, Seattle, WA 9816. *Chapel Language Specification, Version 0.8*, April 2011. Available from: http://chapel.cray.com/spec/spec-0.8.pdf.

[CRM91] Stuart K. Card, George G. Robertson, and Jock D. Mackinlay. The information visualizer, an information workspace. In *Proceedings of the SIGCHI conference on Human factors in computing systems: Reaching through technology*, CHI '91, pages 181–186. ACM, New York, NY, USA, 1991. ISBN 0-89791-383-3. doi:10.1145/108844.108874.

[CSS+91] David E. Culler, Anurag Sah, Klaus E. Schauser, Thorsten von Eicken, and John Wawrzynek. Fine-grain parallelism with minimal hardware support: a compiler-controlled threaded abstract machine. In *ASPLOS-IV: Proc. 4th international conference on Architectural support for programming languages and operating systems*, pages 164–175. ACM, New York, NY, USA, 1991. ISBN 0-89791-380-9. doi:10.1145/106972.106990.

[Dac06] Scott G. Dacko. Narrowing the skills gap for marketers of the future. *Marketing Intelligence & Planning*, 24(3):283–295, 2006. ISSN 0263-4503. doi:10.1108/02634500610665736.

[Dav65] Martin Davis, editor. *The Undecidable, Basic Papers on Undecidable Propositions, Unsolvable Problems And Computable Functions*. Raven Press, Hewlett, N.Y. (USA), 1965. ISBN 0-486-43228-9.

[Day11] Edgar G. Daylight. Dijkstra's rallying cry for generalization: The advent of the recursive procedure, late 1950s, early 1960s. *The Computer Journal*, 2011. Available from: http://comjnl.oxfordjournals.org/content/early/2011/03/08/comjnl.bxr002.abstract, doi:10.1093/comjnl/bxr002.

[DGB+03] James C. Dehnert, Brian K. Grant, John P. Banning, Richard Johnson, Thomas Kistler, Alexander Klaiber, and Jim Mattson. The Transmeta Code Morphing™ software: using speculation, recovery, and adaptive retranslation to address real-life challenges. In *Proc. 1st international symposium on Code generation and optimization: feedback-directed and runtime optimization*, CGO '03, pages 15–24. IEEE Computer Society, Washington, DC, USA, 2003. ISBN 0-7695-1913-X.

[DGH+08] Jack Dongarra, Robert Graybill, William Harrod, Robert Lucas, Ewing Lusk, Piotr Luszczek, Janice Mcmahon, Allan Snavely, Jeffrey Vetter, Katherine Yelick, Sadaf Alam, Roy Campbell, Laura Carrington, Tzu-Yi Chen, Omid Khalili, Jeremy Meredith, and Mustafa Tikir. DARPA's HPCS program: History, models, tools, languages. In Marvin V. Zelkowitz, editor, *Advances in COMPUTERS High Performance Computing*, volume 72 of *Advances in Computers*, pages 1–100. Elsevier, 2008. ISSN 0065-2458. doi:10.1016/S0065-2458(08)00001-6.

[Dij71] E. W. Dijkstra. Hierarchical ordering of sequential processes. *Acta Informatica*, 1(2):115–138, June 1971. ISSN 0001-5903. doi:10.1007/BF00289519.

[DKK+12] M. Daněk, L. Kafka, L. Kohout, J. Sýkora, and R. Bartosinski. *UTLEON3: Exploring Fine-Grain Multi-Threading in FPGAs*. Circuits and Systems. Springer, November 2012. ISBN 978-1-4614-2409-3. Available from: http://www.springer.com/engineering/circuits+%26+systems/book/978-1-4614-2409-3.

[DKKS10] M. Danek, L. Kafka, L. Kohout, and J. Sykora. Instruction set extensions for multi-threading in LEON3. In Z. Kotasek et al., editor, *Proc. 13th IEEE Symposium on Design and Diagnostics of Electronic Circuits and Systems (DDECS'2010)*, pages 237–242. IEEE, 2010. ISBN 978-1-4244-6610-8. doi:10.1109/DDECS.2010.5491777.

[Duf09] Joe Duffy. *Concurrent Programming on Windows*, chapter Task Parallel Library, pages 887—929. Microsoft .NET development series. Addison-Wesley, Boston, MA, USA, 2009. ISBN 032143482X.

[DZJ08] T. D.Vu, L. Zhang, and C. R. Jesshope. The verification of the on-chip COMA cache coherence protocol. In *International Conference on Algebraic Methodology and Software Technology*, pages 413–429, 2008. ISBN 978-3-540-79979-5.

[EBSA+11] Hadi Esmaeilzadeh, Emily Blem, Renee St. Amant, Karthikeyan Sankaralingam, and Doug Burger. Dark silicon and the end of multicore scaling. In *Proc. 38th annual international symposium on Computer architecture*, ISCA '11, pages 365–376. ACM, New York, NY, USA, 2011. ISBN 978-1-4503-0472-6. doi:10.1145/2000064.2000108.

[FE81] M. Feller and M. Ercegovac. Queue machines: An organization for parallel computation. In W. Brauer, P. Hansen, D. Gries, C. Moler, G. Seegmüller, J. Stoer, N. Wirth, and Wolfgang Händler, editors, *Conpar 81*, volume 111 of *Lecture Notes in Computer Science*, pages 37–47. Springer Berlin / Heidelberg, 1981. doi:10.1007/BFb0105108.

[FH76] W. S. Ford and V. C. Hamacher. Hardware support for inter-process communication and processor sharing. *SIGARCH Comput. Archit. News*, 4(4):113–118, January 1976. ISSN 0163-5964. doi:10.1145/633617.803559.

[Fie00] Roy Thomas Fielding. *Architectural Styles and the Design of Network-based Software Architectures*. PhD thesis, University of California, Irvine, 2000. Available from: http://www.ics.uci.edu/~fielding/pubs/dissertation/fielding_dissertation.pdf.

[FLR98] Matteo Frigo, Charles E. Leiserson, and Keith H. Randall. The implementation of the Cilk-5 multithreaded language. *SIGPLAN Not.*, 33:212–223, May 1998. ISSN 0362-1340. doi:10.1145/277652.277725.

[Foua] Free Software Foundation. Alpha relocations, section 9.1.3.3 of the GNU assembler user guide. Available from: http://sourceware.org/binutils/docs/as/Alpha_002dRelocs.html [cited September 2011].

[Foub] Python Software Foundation. What is Python? Executive Summary. http://www.python.org/doc/essays/blurb/.

[Frea] Free Software Foundation. Assembler instructions with C expression operands, section 6.41 of the GNU compiler collection documentation. Available from: http://gcc.gnu.org/onlinedocs/gcc/Extended-Asm.html [cited September 2011].

[Freb] Free Software Foundation. GNU C library. Available from: http://www.gnu.org/s/libc/ [cited October 2011].

[Frec] Free Software Foundation. Options for code generation conventions, section 3.18 of the GNU compiler collection documentation. Available from: http://gcc.gnu.org/onlinedocs/gcc/Code-Gen-Options.html [cited September 2011].

[Fred] Free Software Foundation. Referring to a type with typeof, section 6.6 of the GNU compiler collection documentation. Available from: http://gcc.gnu.org/onlinedocs/gcc/Typeof.html [cited September 2011].

[Free] Free Software Foundation. Specifying registers for local variables, section 6.44.2 of the GNU compiler collection documentation. Available from: http://gcc.gnu.org/onlinedocs/gcc/Local-Reg-Vars.html [cited September 2011].

[GA04] C. Gacek and B. Arief. The many meanings of open source. *IEEE Software*, 21(1):34–40, January-February 2004. ISSN 0740-7459. doi:10.1109/MS.2004.1259206.

[GAB+88] R.B. Garner, A. Agrawal, F. Briggs, E.W. Brown, D. Hough, B. Joy, S. Kleiman, S. Muchnick, M. Namjoo, D. Patterson, J. Pendleton, and R. Tuck. The scalable processor architecture (SPARC). In *Compcon Spring '88. 33rd IEEE Computer Society International Conference, Digest of Papers*, pages 278–283. IEEE, February/March 1988. doi:10.1109/CMPCON.1988.4874.

[Gar70] Martin Gardner. Mathematical games: The fantastic combinations of John Conway's new solitaire game "life". *Scientific American*, 223:120–123, October 1970. ISBN 0894540017. Available from: http://ddi.cs.uni-potsdam.de/HyFISCH/Produzieren/lis_projekt/proj_gamelife/ConwayScientificAmerican.htm.

[GBK+09] Z. Guz, E. Bolotin, I. Keidar, A. Kolodny, A. Mendelson, and U.C. Weiser. Many-core vs. many-thread machines: Stay away from the valley. *Computer Architecture Letters*, 8(1):25–28, January 2009. ISSN 1556-6056. doi:10.1109/L-CA.2009.4.

[GEMN07] M. Gschwind, D. Erb, S. Manning, and M. Nutter. An open source environment for cell broadband engine system software. *Computer*, 40(6):37–47, june 2007. ISSN 0018-9162. doi:10.1109/MC.2007.192.

[GJWJ07] B. Ganesh, A. Jaleel, D. Wang, and B. Jacob. Fully-buffered DIMM memory architectures: Understanding mechanisms, overheads and scaling. In *Proc. IEEE 13th International Symposium on High Performance Computer Architecture (HPCA'2007)*, pages 109–120, February 2007. doi:10.1109/HPCA.2007.346190.

[Goo] Google Inc. The Go programming language: Language design FAQ. http://golang.org/doc/go_lang_faq.html, retrieved 2010-03-26.

[Gru94] Dirk Grunwald. Heaps o' stacks: Time and space efficient threads without operating system support. Technical Report A099254 (CU-CS-750-94), University of Colorado at Boulder—Department of Computer Science, November 1994. Available from: http://handle.dtic.mil/100.2/ADA452990.

[GS00] J. Gray and P. Shenoy. Rules of thumb in data engineering. In *Proc. 16th International Conference on Data Engineering.*, pages 3–10. IEEE, 2000. doi:10.1109/ICDE.2000.839382.

[GS06] Clemens Grelck and Sven-Bodo Scholz. SAC: a functional array language for efficient multithreaded execution. *International Journal of Parallel Programming*, 34(4):383–427, Aug 2006.

[GSS10] Clemens Grelck, Sven-Bodo Scholz, and Alex Shafarenko. Asynchronous stream processing with S-Net. *International Journal of Parallel Programming*, 38(1):38–67, 2010. doi:10.1007/s10766-009-0121-x.

[Gus88] John L. Gustafson. Reevaluating amdahl's law. *Commun. ACM*, 31(5):532–533, May 1988. ISSN 0001-0782. doi:10.1145/42411.42415.

[GVTP97] Antonio González, Mateo Valero, Nigel Topham, and Joan M. Parcerisa. Eliminating cache conflict misses through XOR-based placement functions. In *Proc. 11th International Conference on Supercomputing (ICS'97)*, pages 76–83. ACM, New York, NY, USA, 1997. ISBN 0-89791-902-5. doi:10.1145/263580.263599.

[GWG80] John Gurd, Ian Watson, and John Glauert. A multilayered data flow computer architecture. Technical Report UMCS-80-3-1, Department of Computer Science, University of Manchester, Oxford Road, Manchester, M13 9PL, England, March 1980. Available from: http://intranet.cs.man.ac.uk/Intranet_subweb/library/cstechrep/Abstracts/UMCS-80-3-1.html.

[HÏ10] Philip Kaj Ferdinand Hölzenspies. *On run-time exploitation of concurrency.* PhD thesis, University of Twente, Enschede, the Netherlands, April 2010. Available from: http://doc.utwente.nl/70959/, doi:10.3990/1.9789036530217.

[Hal70] Daniel Stephen Halacy. *Charles Babbage, father of the computer.* Crowell-Collier Press, 1970. ISBN 0027413705.

[Hal85] Robert H. Halstead, Jr. Multilisp: a language for concurrent symbolic computation. *ACM Trans. Program. Lang. Syst.*, 7(4):501–538, 1985. ISSN 0164-0925. doi:10.1145/4472.4478.

[Har11] G.H. Hardy. The new symbolic logic (review of Whitehead's & Russell's Principia Mathematica, Vol. I). *The Times Literary Supplement*, (504):321–322, September 1911. Available from: http://www.cl.cam.ac.uk/~ns441/files/hardy-principia.pdf.

[Has06] Nabil M. Hasasneh. *Chip Multi-Processors using a Micro-Threaded model.* PhD thesis, University of Hull, October 2006.

[HBJ07] Nabil Hasasneh, Ian Bell, and Chris Jesshope. Asynchronous arbiter for micro-threaded chip multiprocessors. *Journal of Systems Architecture*, 53(5-6):253–262, 2007. ISSN 1383-7621. Available from: http://www.sciencedirect.com/science/article/pii/S1383762106001263, doi:10.1016/j.sysarc.2006.10.004.

[HBS73] Carl Hewitt, Peter Bishop, and Richard Steiger. A universal modular ACTOR formalism for artificial intelligence. In *Proc. 3rd International Joint Conference on Artificial Intelligence*, pages 235–245. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1973.

[HBT07] Michael Haller, Mark Billinghurst, and Bruce Thomas, editors. *Emerging Technologies of Augmented Reality: Interfaces and Design*. IGI Global, 2007. ISBN 9781599040660. doi:10.4018/978-1-59904-066-0.

[HEV+98] Gernot Heiser, Kevin Elphinstone, Jerry Vochteloo, Stephen Russell, and Jochen Liedtke. The mungi single-address-space operating system. *Software: Practice and Experience*, 28 (9):901–928, 1998. ISSN 1097-024X. doi:10.1002/(SICI)1097-024X(19980725)28:9<901::AID-SPE181>3.0.CO;2-7.

[HF88] R. H. Halstead, Jr. and T. Fujita. MASA: a multithreaded processor architecture for parallel symbolic computing. *SIGARCH Comput. Archit. News*, 16:443–451, May 1988. ISSN 0163-5964. doi:10.1145/633625.52449.

[HJS+11] Stephan Herhut, Carl Joslin, Sven-Bodo Scholz, Raphael Poss, and Clemens Grelck. Concurrent non-deferred reference counting on the Microgrid: First experiences. In J. Haage and M. Morazán, editors, *Implementation and Application of Functional Languages*, volume 6647 of *Lecture Notes in Computer Science*, pages 185–202. Springer Berlin / Heidelberg, 2011. ISBN 978-3-642-24275-5. doi:10.1007/978-3-642-24276-2_12.

[Hoa78] C. A. R. Hoare. Communicating sequential processes. *Commun. ACM*, 21:666–677, August 1978. ISSN 0001-0782. doi:10.1145/359576.359585.

[Hoa85] C. A. R. Hoare. *Communicating sequential processes*. Prentice Hall, 1985. ISBN 0-13-153289-8. Available from: http://www.usingcsp.com/cspbook.pdf.

[Hof99] Douglas R. Hofstadter. *Gödel, Escher, Bach: An Eternal Golden Braid*. Basic Books; 20 Anv edition, February 1999. ISBN 978-0465026562.

[Hol89] Herman Hollerith. *An Electric Tabulating System*. PhD thesis, Columbia University, 1889. Available from: http://www.columbia.edu/cu/computinghistory/tabulator.html.

[HV05] Jon Haas and Pete Vogt. Fully-buffered DIMM technology moves enterprise platforms to the next level. *Technology@Intel Magazine*, March 2005.

[HWC04] J. Henkel, W. Wolf, and S. Chakradhar. On-chip networks: a scalable, communication-centric embedded system design paradigm. In *Proc. 17th International Conference on VLSI Design*, pages 845–851, 2004. doi:10.1109/ICVD.2004.1261037.

[IBMa] IBM. XL C/C++ v8.0 for AIX: Extensions for GNU C compatibility. Available from: http://publib.boulder.ibm.com/infocenter/comphelp/v8v101/topic/com.ibm.xlcpp8a.doc/language/ref/gcc_cext.htm#gcc_cext [cited September 2011].

[IBMb] IBM. XL C/C++ v8.0 for AIX: The typeof operator. Available from: http://publib.boulder.ibm.com/infocenter/comphelp/v8v101/topic/com.ibm.xlcpp8a.doc/language/ref/typeof_operator.htm [cited September 2011].

[IEE08] IEEE Standards Association. *IEEE Std. 1003.1-2008, Information Technology – Portable Operating System Interface (POSIX®)*. IEEE, 2008.

[II99] International Standards Organization and International Electrotechnical Commission. *ISO/IEC 9899:1999(E), Programming Languages – C*. American National Standards Institute (ANSI), 11 West 42nd Street, New York, New York 1O036, second edition, December 1999.

[II03] International Standards Organization and International Electrotechnical Commission. *ISO/IEC 14882:2003, Programming languages – C++*. American National Standards Institute (ANSI), 11 West 42nd Street, New York, New York 1O036, second edition, October 2003. Available from: http://www.open-std.org/jtc1/sc22/wg21/.

[II09] International Standards Organization and International Electrotechnical Commission. *ISO/IEC 9945:2009, Information technology – Portable Operating System Interface (POSIX®) Base Specifications, Issue 7*. American National Standards Institute (ANSI), 11 West 42nd Street, New York, New York 1O036, 2009.

[II11a] International Standards Organization and International Electrotechnical Commission. *ISO/IEC 14882:2011, Programming languages – C++*. American National Standards Institute (ANSI), 11 West 42nd Street, New York, New York 1O036, first edition, September 2011. Available from: `http://www.open-std.org/jtc1/sc22/wg21/`.

[II11b] International Standards Organization and International Electrotechnical Commission. *ISO/IEC 9899:2011, Programming Languages – C*. American National Standards Institute (ANSI), 11 West 42nd Street, New York, New York 1O036, first edition, December 2011. Available from: `http://www.open-std.org/jtc1/sc22/wg14/`.

[Ili61] J.K. Iliffe. The use of the genie system in numerical calculation. *Annual Review in Automatic Programming*, 2:1–28, 1961. ISSN 0066-4138. Available from: `http://www.sciencedirect.com/science/article/pii/S0066413861800025`, `doi:10.1016/S0066-4138(61)80002-5`.

[Ins95] Institute of Electrical and Electronic Engineers, Inc. Information Technology — Portable Operating Systems Interface (POSIX) — Part: System Application Program Interface (API) — Amendment 2: Threads Extension [C Language]. IEEE Standard 1003.1c–1995, IEEE, New York City, New York, USA, 1995. also ISO/IEC 9945-1:1990b.

[Inta] Intel Corporation. Intel® C++ compiler options. Available from: `http://software.intel.com/sites/products/documentation/hpc/compilerpro/en-us/cpp/mac/man/icc.txt` [cited September 2011].

[Intb] Intel Corporation. Intel® C++ compiler user and reference guides: gcc* compatibility. Available from: `http://software.intel.com/sites/products/documentation/hpc/compilerpro/en-us/cpp/lin/compiler_c/bldaps_cls/common/bldaps_gcc_compat_comm.htm` [cited September 2011].

[Intc] Intel Corporation. Intel® C++ compiler user and reference guides: Intrinsics reference: Inline assembly. Available from: `http://software.intel.com/sites/products/documentation/hpc/compilerpro/en-us/cpp/lin/compiler_c/intref_cls/common/intref_data_align_ma_ia_linux_ia.htm` [cited September 2011].

[Int11] Intel Corporation. *Intel® Threading Building Blocks Reference Manual*, 2011. Available from: `http://threadingbuildingblocks.org/documentation.php`.

[Jes01] C. Jesshope. Implementing an efficient vector instruction set in a chip multi-processor using micro-threaded pipelines. In *Proc. 6th Australasian Computer Systems Architecture Conference (ACSAC'01)*, pages 80–88. IEEE, 2001. ISBN 0-7695-0954-1. `doi:10.1109/ACAC.2001.903363`.

[Jes03] Chris Jesshope. Multi-threaded microprocessors – evolution or revolution. In Amos Omondi and Stanislav Sedukhin, editors, *Advances in Computer Systems Architecture*, volume 2823 of *Lecture Notes in Computer Science*, pages 21–45. Springer Berlin / Heidelberg, 2003. `doi:10.1007/978-3-540-39864-6_4`.

[Jes04] Chris Jesshope. Scalable instruction-level parallelism. In Andy Pimentel and Stamatis Vassiliadis, editors, *Computer Systems: Architectures, Modeling, and Simulation*, volume 3133 of *Lecture Notes in Computer Science*, pages 383–392. Springer Berlin / Heidelberg, 2004. `doi:10.1007/978-3-540-27776-7_40`.

[Jes05] C.R. Jesshope. Microgrids – the exploitation of massive on-chip concurrency. In Lucio Grandinetti, editor, *Grid Computing The New Frontier of High Performance Computing*, volume 14 of *Advances in Parallel Computing*, pages 203–223. North-Holland, 2005. ISSN 0927-5452. Available from: `http://www.sciencedirect.com/science/article/pii/S0927545205800127`, `doi:10.1016/S0927-5452(05)80012-7`.

[Jes06a] Chris R. Jesshope. μTC - an intermediate language for programming chip multiprocessors. In Chris Jesshope and Colin Egan, editors, *Advances in Computer Systems Architecture*, volume 4186 of *Lecture Notes in Computer Science*, pages 147–160. Springer Berlin / Heidelberg, 2006. ISBN 978-3-540-40056-1. `doi:10.1007/11859802_13`.

[Jes06b] C.R. Jesshope. Microthreading, a model for distributed instruction-level concurrency. *Parallel Processing Letters*, 16(2):209–228, 2006.

[Jes08a] Chris Jesshope. Operating systems in silicon and the dynamic management of resources in many-core chips. *Parallel Processing Letters*, 18(2):257–274, 2008.

[Jes08b] C.R. Jesshope. A model for the design and programming of multi-cores. In Lucio Grandinetti, editor, *High Performance Computing and Grids in Action*, number 16 in Advances in Parallel Computing, pages 37–55. IOS Press, 2008. ISBN 978-1-58603-839-7. Available from: `http://dare.uva.nl/record/288698`.

[JIS] JISC. OpenIndiana. Available from: `http://openindiana.org/` [cited October 2011].

[JL00] C. Jesshope and B. Luo. Micro-threading: a new approach to future RISC. In *Proc. 5th Australasian Computer Architecture Conference (ACAC'00)*, pages 34–41. IEEE, 2000. ISBN 0-7695-0512-0. `doi:10.1109/ACAC.2000.824320`.

[JLI98] Trent Jaeger, Jochen Liedtke, and Nayeem Islam. Operating system protection for fine-grained programs. In *Proc. 7th conference on USENIX Security Symposium - Volume 7*, page 11. USENIX Association, Berkeley, CA, USA, 1998. Available from: `http://usenix.net/publications/library/proceedings/sec98/full_papers/jaeger/jaeger.pdf`.

[JLZ09a] Chris Jesshope, Mike Lankamp, and Li Zhang. Evaluating CMPs and their memory architecture. In *Architecture of Computing Systems – ARCS 2009*, volume 5455/2009 of *Lecture Notes in Computer Science*, pages 246–257. Springer Berlin / Heidelberg, 2009. ISBN 978-3-642-00453-7. ISSN 0302-9743 (Print) 1611-3349 (Online). `doi:10.1007/978-3-642-00454-4_24`.

[JLZ09b] Chris Jesshope, Mike Lankamp, and Li Zhang. The Implementation of an SVP Many-core Processor and the Evaluation of its Memory Architecture. *ACM SIGARCH Computer Architecture News*, 37(2):38–45, 2009. ISSN 0163-5964. `doi:10.1145/1577129.1577136`.

[JMW+03] Adrian Jacobs, Jonathan Mather, Robert Winlow, David Montgomery, Graham Jones, Morgan Willis, Martin Tillin, Lyndon Hill, Marina Khazova, Heather Stevenson, and Grant Bourhill. 2D/3D switchable displays. *Sharp Technical Journal*, 4, 2003. Available from: `http://www.sharp.co.jp/corporate/rd/journal-85/pdf/85-04.pdf`.

[Joe96] Christopher F. Joerg. *The Cilk system for parallel multithreaded computing.* PhD thesis, Massachusetts Institute of Technology, Cambridge, MA, USA, 1996.

[JPvT08] Chris Jesshope, Jean-Marc Philippe, and Michiel van Tol. An architecture and protocol for the management of resources in ubiquitous and heterogeneous systems based on the SVP model of concurrency. In *Embedded Computer Systems: Architectures, Modeling, and Simulation*, pages 218–228, 2008. ISBN 978-3-540-70549-9.

[JR81] S. C. Johnson and D. M. Ritchie. The C language calling sequence. Computing Science Technical Report 102, Bell Laboratories, September 1981. Available from: `http://cm.bell-labs.com/cm/cs/who/dmr/clcs.ps`.

[JS08] Chris Jesshope and Alex Shafarenko. Concurrency Engineering. In *Proc. 13th IEEE Asia-Pacific Computer Systems Architecture Conference*, 2008. ISBN 978-1-4244-2683-6.

[JYS+12] Min Kyu Jeong, Doe Hyun Yoon, Dam Sunwoo, Mike Sullivan, Ikhwan Lee, and Mattan Erez. Balancing DRAM locality and parallelism in shared memory CMP systems. In *Proc. 18th International Symposium on High Performance Computer Architecture*, HPCA'12, pages 1–12. IEEE, February 2012. ISSN 1530-0897. `doi:10.1109/HPCA.2012.6168944`.

[KDH+05] J. A. Kahle, M. N. Day, H. P. Hofstee, C. R. Johns, T. R. Maeurer, and D. Shippy. Introduction to the Cell multiprocessor. *IBM Journal of Research and Development*, 49(4.5):589–604, July 2005. ISSN 0018-8646. `doi:10.1147/rd.494.0589`.

[Ken08] Randall C. Kennedy. Fat, fatter, fattest: Microsoft's kings of bloat. *InfoWorld*, April 2008. Available from: `http://www.infoworld.com/t/applications/fat-fatter-fattest-microsofts-kings-bloat-278`.

[Khr09] Khronos OpenCL Working Group. The OpenCL specification, version 1.0.43, 2009. Available from: `http://www.khronos.org/registry/cl/specs/opencl-1.0.43.pdf`.

[Kic91] G. Kiczales. Towards a new model of abstraction in software engineering. In *Proc. 1991 International Workshop on Object Orientation in Operating Systems*, pages 127–128. IEEE, October 1991. `doi:10.1109/IWOOOS.1991.183036`.

[Kis02] Laszlo B. Kish. End of moore's law: thermal (noise) death of integration in micro and nano electronics. *Physics Letters A*, 305(3–4):144–149, 2002. ISSN 0375-9601. doi:10.1016/S0375-9601(02)01365-8.

[KISL04] Mazen Kharbutli, Keith Irwin, Yan Solihin, and Jaejin Lee. Using prime numbers for cache indexing to eliminate conflict misses. In *Proceedings of the 10th International Symposium on High Performance Computer Architecture (HPCA'04)*, pages 288–299. IEEE Computer Society, Washington, DC, USA, 2004. ISBN 0-7695-2053-7. doi:10.1109/HPCA.2004.10015.

[KJS+02] S. Kumar, A. Jantsch, J.-P. Soininen, M. Forsell, M. Millberg, J. Oberg, K. Tiensyrja, and A. Hemani. A network on chip architecture and design methodology. In *Proc. IEEE Computer Society Annual Symposium on VLSI*, pages 105–112, 2002. doi:10.1109/ISVLSI.2002.1016885.

[Kla00] Alexander Klaiber. The technology behind Crusoe processors. Transmeta Corporation White Paper, January 2000. Available from: http://scholar.google.nl/scholar?cluster=16353518713752049801.

[KMZS08] Henry Kasim, Verdi March, Rita Zhang, and Simon See. Survey on parallel programming model. In *Network and Parallel Computing*, volume 5245/2008 of *Lecture Notes in Computer Science*, pages 266–275. Springer Berlin / Heidelberg, 2008.

[Knu98] Donald Knuth. *The Art of Computer Programming*, volume 3: Sorting and Searching. Addison-Wesley, third edition, 1998. ISBN 0-201-89685-0.

[Kon07] Petr Konecny. Introducing the Cray XMT. In *Proc. Cray User Group meeting (CUG'07)*. Cray Inc., 411 First Avenue South, Seattle, WA 9810, USA, May 2007.

[KR77] Brian W. Kernighan and Dennis M. Ritchie. *The M4 macro processor*. AT&T Bell Laboratories, 1977.

[Lan07] M. Lankamp. Developing a Reference Implementation for a Microgrid of Microthreaded Microprocessors. Master's thesis, University of Amsterdam, Amsterdam, the Netherlands, August 2007. Available from: http://dist.svp-home.org/doc/mike-lankamp-ref-microgrid.pdf.

[Lan1x] Mike Lankamp. *Design and Evaluation of a Multithreaded Many-Core Architecture*. PhD thesis, University of Amsterdam, 201x. To appear.

[LB08] Mike Lankamp and Thomas Bernard. Microthreads reference document (draft). Technical report, University of Amsterdam, November 2008.

[Lea96] Doug Lea. A memory allocator, 1996. Available from: http://g.oswego.edu/dl/html/malloc.html [cited September 2011].

[Lea10] Richard Leadbetter. Digital Foundry vs. 3DS. *Eurogamer*, June 2010. Available from: http://www.eurogamer.net/articles/digitalfoundry-vs-nintendo-3ds.

[Lei09] Charles E. Leiserson. The Cilk++ concurrency platform. In *DAC '09: Proceedings of the 46th Annual Design Automation Conference*, pages 522–527. ACM, New York, NY, USA, 2009. ISBN 978-1-60558-497-3. doi:10.1145/1629911.1630048.

[Lev84] Steven Levy. *Hackers: Heroes of the computer revolution*. Anchor Press/Doubleday, 1984. ISBN 0385191952.

[LF80] Richard E. Ladner and Michael J. Fischer. Parallel prefix computation. *J. ACM*, 27(4): 831–838, October 1980. ISSN 0004-5411. doi:10.1145/322217.322232.

[LH94] B. Lee and A.R. Hurson. Dataflow architectures and multithreading. *IEEE Computer*, 27(8): 27–39, aug. 1994. ISSN 0018-9162. doi:10.1109/2.303620.

[Lin76] Theodore A. Linden. Operating system structures to support security and reliable software. *ACM Comput. Surv.*, 8:409–445, December 1976. ISSN 0360-0300. doi:10.1145/356678.356682.

[LJ02] Bing Luo and Chris Jesshope. Performance of a micro-threaded pipeline. In *Proc. 7th Asia-Pacific conference on Computer systems architecture (CRPIT '02)*, pages 83–90. Australian Computer Society, Inc., Darlinghurst, Australia, Australia, 2002. ISBN 0-909925-84-4.

[LLV] LLVM Developers. Clang information: Language compatibility: Inline assembly. Available from: http://clang.llvm.org/compatibility.html#inline-asm [cited September 2011].

[LNOM08] E. Lindholm, J. Nickolls, S. Oberman, and J. Montrym. NVIDIA Tesla: A unified graphics and computing architecture. *IEEE Micro*, 28(2):39–55, March/April 2008. ISSN 0272-1732. doi:10.1109/MM.2008.31.

[LT02] Josh Lerner and Jean Tirole. Some simple economics of open source. *The Journal of Industrial Economics*, 50(2):197–234, June 2002. ISSN 1467-6451. doi:10.1111/1467-6451.00174.

[LT04] Josh Lerner and Jean Tirole. The economics of technology sharing: Open source and beyond. Working Paper 10956, National Bureau of Economic Research, 1050 Massachusetts Avenue, Cambridge, MA 02138, USA, December 2004. Available from: http://www.nber.org/papers/w10956.

[LW03] Karim Lakhani and Robert G. Wolf. Why hackers do what they do: Understanding motivation and effort in free/open source software projects. MIT Sloan Working Paper 4425-03, Massachusetts Institute of Technology, MIT Sloan School of Management, E60-186, 30 Memorial Drive, Cambridge, MA 02142, USA, September 2003. Available from: http://ssrn.com/abstract=443040, doi:10.2139/ssrn.443040.

[MAG+88] S.S. Muchnick, C. Aoki, V. Ghodssi, M. Helft, M. Lee, R. Tuck, D. Weaver, and A. Wu. Optimizing compilers for the SPARC architecture – an overview. In *Compcon Spring '88. Thirty-Third IEEE Computer Society International Conference, Digest of Papers*, pages 284–288. IEEE, February/March 1988. doi:10.1109/CMPCON.1988.4875.

[Mat10] Andrei Matei. Towards adaptable parallel software - the Hydra runtime for SVP programs. Master's thesis, Vrije Universiteit Amsterdam, Faculty of Science, Department of Mathematics and Computer Science, November 2010. Available from: http://dist.svp-home.org/doc/andrei-matei-hydra-svp-runtime.pdf.

[MBH+02] D. T. Marr, Frank Binns, D. L. Hill, Glenn Hinton, D. A. Koufaty, J. A. Miller, and Michael Upton. Hyper-Threading technology architecture and microarchitecture. *Intel Technology Journal*, 6(1):1–12, 2002. Available from: http://www.mendeley.com/research/hyperthreading-technology-architecture-and-microarchitecture/.

[MBKQ96] Marshall K. McKusick, Keith Bostic, Michael J. Karels, and John S. Quartermain. *The Design and Implementation of the 4.4BSD Operating System*. Addison-Wesley, 1996. ISBN 978-0-201-54979-9.

[MBRS11] Jeffrey C. Mogul, Andrew Baumann, Timothy Roscoe, and Livio Soares. Mind the gap: reconnecting architecture and OS research. In *Proc. 13th USENIX conference on Hot topics in operating systems*, HotOS'13. USENIX Association, Berkeley, CA, USA, 2011.

[McK99] Marshall K. McKusick. *Open Sources: Voices from the Open Source Revolution*, chapter Twenty Years of Berkeley Unix - From AT&T-Owned to Freely Redistributable. O'Reilly & Associates, 1999. ISBN 978-1-565-92582-3. Available from: http://oreilly.com/catalog/opensources/book/kirkmck.html.

[McM86] F.H. McMahon. The livermore FORTRAN kernels: A computer test of the numerical performance range. Technical Report UCRL-53745, Lawrence Livermore National Lab., CA (USA), Dec 1986.

[MCS91] John M. Mellor-Crummey and Michael L. Scott. Synchronization without contention. *SIGPLAN Not.*, 26(4):269–278, 1991. ISSN 0362-1340. doi:10.1145/106973.106999.

[Mes09] Message Passing Interface Forum. *MPI: A Message-Passing Interface Standard, Version 2.2*. High Performance Computing Center Stuttgart (HLRS), September 2009.

[Mil68] Robert B. Miller. Response time in man-computer conversational transactions. In *Proceedings of the December 9-11, 1968, fall joint computer conference, part I*, AFIPS '68 (Fall, part I), pages 267–277. ACM, New York, NY, USA, 1968. doi:10.1145/1476589.1476628.

[Mil80] Robin Milner. *A Calculus of Communicating Systems*. Springer-Verlag New York, Inc., 1980. ISBN 0-387-10235-3.

[Mil90] Robin Milner. Functions as processes. In Michael Paterson, editor, *Automata, Languages and Programming*, volume 443 of *Lecture Notes in Computer Science*, pages 167–180. Springer Berlin / Heidelberg, 1990. ISBN 978-3-540-52826-5. doi:10.1007/BFb0032030.

[MMB07] Orlando Moreira, Jacob Jan-David Mol, and Marco Bekooij. Online resource management in a multiprocessor with a network-on-chip. In *Proc. 2007 ACM symposium on Applied computing*, SAC '07, pages 1557–1564. ACM, 2007. ISBN 1-59593-480-4. doi:10.1145/1244002.1244335.

[MNN04] Marshall Kirk McKusick and George V Neville-Neil. *Design And Implementation Of The FreeBSD Operating System*. Addison Wesley, 2004. 720 pp.

[Mor66] G.M. Morton. A computer oriented geodetic data base; and a new technique in file sequencing. Technical report, International Business Machines Ltd., Ottawa, Canada, 1966.

[MPW92a] Robin Milner, Joachim Parrow, and David Walker. A calculus of mobile processes, I. *Information and Computation*, 100(1):1–40, 1992. ISSN 0890-5401. doi:10.1016/0890-5401(92)90008-4.

[MPW92b] Robin Milner, Joachim Parrow, and David Walker. A calculus of mobile processes, II. *Information and Computation*, 100(1):41–77, 1992. ISSN 0890-5401. doi:10.1016/0890-5401(92)90009-5.

[MRL+10] Timothy G. Mattson, Michael Riepen, Thomas Lehnig, Paul Brett, Werner Haas, Patrick Kennedy, Jason Howard, Sriram Vangal, Nitin Borkar, Greg Ruhl, and Saurabh Dighe. The 48-core scc processor: the programmer's view. In *Proc. 2010 ACM/IEEE International Conference for High Performance Computing, Networking, Storage and Analysis*, SC'10, pages 1–11. IEEE Computer Society, Washington, DC, USA, 2010. ISBN 978-1-4244-7559-9. doi:10.1109/SC.2010.53.

[MS90] David May and Roger Shepherd. Occam and the transputer. In *Advances in Petri Nets 1989*, volume 424 of *Lecture Notes in Computer Science*, pages 329–353. Springer Berlin / Heidelberg, 1990. ISBN 978-3-540-52494-6. ISSN 0302-9743. doi:10.1007/3-540-52494-0.

[MS09] Jiayuan Meng and K. Skadron. Avoiding cache thrashing due to private data placement in last-level cache for manycore scaling. In *IEEE International Conference on Computer Design (ICCD'09)*, pages 282–288. IEEE, October 2009. ISBN 978-1-4244-5029-9. ISSN 1063-6404. doi:10.1109/ICCD.2009.5413143.

[MSA+83] J. McGraw, S. Skedzielewski, S. Allan, D. Grit, R. Oldehoeft, J. Glauert, I. Dobes, and P. Hohensee. SISAL: streams and iteration in a single-assignment language. language reference manual, version 1.2, revision 1. Technical Report LLL/M-146-Rev.1, Lawrence Livermore National Lab., CA (USA); Colorado State Univ., Fort Collins (USA). Dept. of Computer Science; East Anglia Univ. (UK). School of Information Sciences; Manchester Univ. (UK). Computer Center; Digital Equipment Corp., Nashua, NH (USA), March 1983. Available from: http://www.osti.gov/energycitations/product.biblio.jsp?osti_id=5065243.

[MSS10] Nick Mitchell, Edith Schonberg, and Gary Sevitsky. Four trends leading to java runtime bloat. *IEEE Software*, 27:56–63, 2010. ISSN 0740-7459. doi:10.1109/MS.2010.7.

[NA89] R. S. Nikhil and Arvind. Can dataflow subsume von Neumann computing? *SIGARCH Comput. Archit. News*, 17(3):262–272, 1989. ISSN 0163-5964. doi:10.1145/74926.74955.

[Nie94] Jakob Nielsen. *Usability Engineering*. Morgan Kaufmann, new edition, November 1994. ISBN 978-0125184069. Available from: http://www.useit.com/papers/responsetime.html.

[NMRW02] George Necula, Scott McPeak, Shree Rahul, and Westley Weimer. Cil: Intermediate language and tools for analysis and transformation of C programs. In R. Horspool, editor, *Compiler Construction*, volume 2304 of *Lecture Notes in Computer Science*, pages 209–265. Springer Berlin / Heidelberg, 2002. doi:10.1007/3-540-45937-5_16.

[NR98] Robert W. Numrich and John Reid. Co-array fortran for parallel programming. *SIGPLAN Fortran Forum*, 17:1–31, August 1998. ISSN 1061-7264. doi:10.1145/289918.289920.

[NVI09] NVIDIA Corporation. NVIDIA's next generation CUDA compute architecture: Fermi. Whitepaper, 2009. Available from: http://www.nvidia.com/content/PDF/fermi_white_papers/NVIDIA_Fermi_Compute_Architecture_Whitepaper.pdf.

[Ope08] OpenMP Architecture Review Board. OpenMP application program interface, version 3.0, 2008. Available from: http://www.openmp.org/mp-documents/spec30.pdf.

[Ora11] Oracle Corporation. *Oracle® Solaris Studio 12.2: C User's Guide*, June 2011. Available from: http://download.oracle.com/docs/cd/E18659_01/pdf/821-1384.pdf.

[PC90] Gregory M. Papadopoulos and David E. Culler. Monsoon: an explicit token-store architecture. *SIGARCH Comput. Archit. News*, 18:82–91, May 1990. ISSN 0163-5964. doi:10.1145/325096.325117.

[Pét34] Rózsa Péter. Über den Zussammenhang der verschiedenen Begriffe der rekursiven Funktion. *Mathematische Annale*, 110:612–632, 1934.

[PF05] Matt Pharr and Randima Fernando, editors. *GPU Gems 2: Programming Techniques for High-Performance Graphics and General-Purpose Computation*. Addison-Wesley Professional, March 2005. ISBN 978-0321335593.

[PHA10] Heidi Pan, Benjamin Hindman, and Krste Asanovic. Composing parallel software efficiently with lithe. In *PLDI '10: Proceedings of the 2010 ACM SIGPLAN conference on Programming language design and implementation*, pages 376–387. ACM, Toronto, Canada, 2010.

[Pig01] Steven Pigeon. *Contributions à la compression de données*. PhD thesis, Université de Montréal, December 2001.

[PJ96] Charles E. Perkins and David B. Johnson. Mobility support in ipv6. In *Proceedings of the 2nd annual international conference on Mobile computing and networking*, MobiCom '96, pages 27–37. ACM, New York, NY, USA, 1996. ISBN 0-89791-872-X. doi:10.1145/236387.236400.

[PJ10] Raphael Poss and Chris Jesshope. Towards scalable implicit communication and synchronization. In *The First Workshop on Advances in Message Passing (AMP'10)*. Toronto, Canada, June 2010. Available from: http://www.cs.rochester.edu/u/cding/amp/papers/pos/Towards%20Scalable%20Implicit%20Communication%20and%20Synchronization.pdf.

[PJCSH87] Simon L. Peyton Jones, Chris Clack, John Salkild, and Mark Hardie. GRIP—a high-performance architecture for parallel graph reduction. In Gilles Kahn, editor, *Functional Programming Languages and Computer Architecture*, volume 274 of *Lecture Notes in Computer Science*, pages 98–112. Springer Berlin / Heidelberg, 1987. ISBN 0-387-18317-5. doi:10.1007/3-540-18317-5_7.

[PLU+12] Raphael Poss, Mike Lankamp, M. Irfan Uddin, Jaroslav Sýkora, and Leoš Kafka. Heterogeneous integration to simplify many-core architecture simulations. In *Proc. 2012 Workshop on Rapid Simulation and Performance Evaluation: Methods and Tools*, RAPIDO '12, pages 17–24. ACM, 2012. ISBN 978-1-4503-1114-4. doi:10.1145/2162131.2162134.

[PLY+12] Raphael Poss, Mike Lankamp, Qiang Yang, Jian Fu, Michiel W. van Tol, and Chris Jesshope. Apple-CORE: Microgrids of SVP cores (invited paper). In *Proc. 15th Euromicro Conference on Digital System Design*. IEEE, Cesme, Izmir, Turkey, September 2012. (to appear).

[Pol99] Fred J. Pollack. New microarchitecture challenges in the coming generations of CMOS process technologies (keynote address). In *Proc. 32nd annual ACM/IEEE international symposium on Microarchitecture*, MICRO 32. IEEE Computer Society, Washington, DC, USA, 1999. ISBN 0-7695-0437-X.

[PR86] Heinz-Otto Peitgen and Peter Richter. *The Beauty of Fractals*. Springer-Verlag, Heidelberg, 1986. ISBN 0-387-15851-0.

[Rad62] Tibor Radó. On non-computable functions. *Bell Systems Technical Journal*, 41(3):877–884, May 1962. Available from: http://www.alcatel-lucent.com/bstj/vol41-1962/articles/bstj41-3-877.pdf.

[Ran82] Brian Randell. *The Origins of digital computers: selected papers*. Texts and monographs in computer science. Springer-Verlag, 3rd edition, 1982. ISBN 9780387113197.

[Rau91] B. Ramakrishna Rau. Pseudo-randomly interleaved memory. *SIGARCH Comput. Archit. News*, 19:74–83, April 1991. ISSN 0163-5964. doi:10.1145/115953.115961.

[Red] Red Hat. Newlib. Available from: http://www.sourceware.org/newlib/ [cited October 2011].

[Rei07] J. Reinders. *Intel Threading Building Blocks: outfitting C++ for multi-core processor parallelism*. O'Reilly Series. O'Reilly, 2007. ISBN 9780596514808. LCCN 2007299881.

[RH90] Ram Raghavan and John P. Hayes. On randomly interleaved memories. In *Proceedings of the 1990 ACM/IEEE conference on Supercomputing*, pages 49–58. IEEE Computer Society Press, Los Alamitos, CA, USA, 1990. ISBN 0-89791-412-0. Available from: `http://portal.acm.org/citation.cfm?id=110382.110396`.

[RH11] Stuart Rance and Ashley Hanna. ITIL® glossary and abbreviations - English. ITIL glossary, IT Infrastructure Library, July 2011. Available from: `http://www.itil-officialsite.com/InternationalActivities/TranslatedGlossaries.aspx`.

[Rie07] D. Riehle. The economic motivation of open source software: Stakeholder perspectives. *Computer*, 40(4):25–32, april 2007. ISSN 0018-9162. `doi:10.1109/MC.2007.147`.

[RML+01] R. Ronen, A. Mendelson, K. Lai, Shih-Lien Lu, F. Pollack, and J.P. Shen. Coming challenges in microarchitecture and architecture. *Proceedings of the IEEE*, 89(3):325–340, mar 2001. ISSN 0018-9219. `doi:10.1109/5.915377`.

[RT74] Dennis M. Ritchie and Ken Thompson. The UNIX time-sharing system. *Commun. ACM*, 17: 365–375, July 1974. ISSN 0001-0782. `doi:10.1145/361011.361061`.

[S.03] Sandeep S. *GCC Inline Assembly HOWTO*, v0.3 edition, March 2003. Available from: `http://www.ibiblio.org/gferg/ldp/GCC-Inline-Assembly-HOWTO.html`.

[SA05] Lawrence Spracklen and Santosh G. Abraham. Chip multithreading: opportunities and challenges. In *Proc 11th International Symposium on High-Performance Computer Architecture*, HPCA'05, pages 248–252. IEEE, February 2005. ISSN 1530-0897. `doi:10.1109/HPCA.2005.10`.

[Sal65] Jerome H. Saltzer. CTSS technical notes. Technical Report MAC-TR-16, Massachusetts Institute of Technology – Project MAC, 1965.

[SCB+98] Allan Snavely, Larry Carter, Jay Boisseau, Amit Majumdar, Kang Su Gatlin, Nick Mitchell, John Feo, and Brian Koblenz. Multi-processor performance on the Tera MTA. In *Supercomputing '98: Proceedings of the 1998 ACM/IEEE conference on Supercomputing*, pages 1–8. IEEE Computer Society, Washington, DC, USA, 1998. ISBN 0-89791-984-X.

[SCS+08] Larry Seiler, Doug Carmean, Eric Sprangle, Tom Forsyth, Michael Abrash, Pradeep Dubey, Stephen Junkins, Adam Lake, Jeremy Sugerman, Robert Cavin, Roger Espasa, Ed Grochowski, Toni Juan, and Pat Hanrahan. Larrabee: a many-core x86 architecture for visual computing. *ACM Trans. Graph.*, 27:18:1–18:15, August 2008. ISSN 0730-0301. `doi:10.1145/1360612.1360617`.

[SEM09] Dimitris Saougkos, Despina Evgenidou, and George Manis. Specifying loop transformations for C2μTC source-ro-source compiler. In *Proc. of 14th Workshop on Compilers for Parallel Computing (CPC'09), Zürich, Switzerland*. IBM Research Center, January 2009.

[Sez93] André Seznec. A case for two-way skewed-associative caches. *SIGARCH Comput. Archit. News*, 21:169–178, May 1993. ISSN 0163-5964. `doi:10.1145/173682.165152`.

[SGJ+12] M. Shah, R. Golla, P. Jordan, G. Grohoski, J. Barreh, J. Brooks, M. Greenberg, G. Levinsky, M. Luttrell, C. Olson, Z. Samoail, M. Smittle, and T. Ziaja. SPARC T4: A dynamically threaded server-on-a-chip. *IEEE Micro*, PP(99):1, 2012. ISSN 0272-1732. `doi:10.1109/MM.2012.1`.

[SGK+85] Russel Sandberg, David Goldberg, Steve Kleiman, Dan Walsh, and Bob Lyon. Design and implementation of the Sun network filesystem. In *Proc. 1985 USENIX Conference*, pages 119–130. Portland, OR, USA, 1985.

[SHJ11] Sven-Bodo Scholz, Stephan Herhut, and Carl Joslin. Data parallelism and functional parallelism on the Microgrid using Single Assignment C. Deliverable 4.4, issue 1.0, Architecture Paradigms and Programming Languages for Efficient programming of multiple COREs (Apple-CORE, EU FP7-215216), July 2011. Available from: `http://www.apple-core.info/deliverables`.

[Shn84] Ben Shneiderman. Response time and display rate in human performance with computers. *ACM Comput. Surv.*, 16:265–285, September 1984. ISSN 0360-0300. `doi:10.1145/2514.2517`.

[SHZO07] Shubhabrata Sengupta, Mark Harris, Yao Zhang, and John D. Owens. Scan primitives for GPU computing. In Timo Aila and Mark Segal, editors, *Graphics Hardware 2007*, pages 97–106. ACM, Augustus 2007.

[Sim00] D. Sima. The design space of register renaming techniques. *Micro, IEEE*, 20(5):70–83, September/October 2000. ISSN 0272-1732. doi:10.1109/40.877952.

[Sir09] John Siracusa. Grand central dispatch. *Ars Technica*, Mac OS X 10.6 Snow Leopard: the Ars Technica review, 2009. Available from: http://arstechnica.com/apple/reviews/2009/08/mac-os-x-10-6.ars/12.

[SJ96] Randy Show and Theodore Johnson. *Distributed operating systems & algorithms*. Addison Wesley, 1996. ISBN 0201498383.

[SKRC10] Konstantin Shvachko, Hairong Kuang, Sanjay Radia, and Robert Chansler. The Hadoop distributed file system. In *Proc. 26th Symposium on Massive Storage Systems and Technologies (MSST'10)*. IEEE Press, Incline Village, Nevada, May 2010. ISBN 978-1-4244-7153-9.

[SLJ+04] Mike Salib, Ling Liao, Richard Jones, Mike Morse, Ansheng Liu, Dean Samara-Rubio, Drew Alduino, and Mario Paniccia. Silicon photonics. *Intel Technology Journal*, 8(2):143–160, May 2004. ISSN 1535-864X.

[SM11] Dimitris Saougkos and George Manis. Run-time scheduling with the C2uTC parallelizing compiler. In *2nd Workshop on Parallel Programming and Run-Time Management Techniques for Many–Core Architectures, in Workshop Proceedings of the 24th Conference on Computing Systems (ARCS 2011)*, Lecture Notes in Computer Science, pages 151–157. Springer, 2011.

[Smi81] B.J. Smith. Architecture and applications of the HEP multiprocessor computer system. *Proc. SPIE Int. Soc. Opt. Eng.; (United States)*, 298:241–248, 1981.

[SPB+08] Adrian Schüpbach, Simon Peter, Andrew Baumann, Timothy Roscoe, Paul Barham, Tim Harris, and Rebecca Isaacs. Embracing diversity in the barrelfish manycore operating system. In *Proceedings of the Workshop on Managed Many-Core Systems*. ACM, June 2008.

[Sut05] Herb Sutter. The free lunch is over: A fundamental turn toward concurrency in software. *Dr. Dobb's Journal*, 30(3), 2005. Available from: http://www.gotw.ca/publications/concurrency-ddj.htm.

[Tab10] Hiroko Tabushi. Nintendo to make 3-D version of its DS handheld game. *The New York Times*, March 2010. Available from: http://www.nytimes.com/2010/03/24/technology/24nintendo.html.

[TEL95] Dean M. Tullsen, Susan J. Eggers, and Henry M. Levy. Simultaneous multithreading: maximizing on-chip parallelism. *SIGARCH Comput. Archit. News*, 23:392–403, May 1995. ISSN 0163-5964. doi:10.1145/225830.224449.

[Tho65] James E. Thornton. Parallel operation in the Control Data 6600. In *Proceedings of the October 27-29, 1964, fall joint computer conference, part II: very high speed computer systems*, AFIPS '64 (Fall, part II), pages 33–40. ACM, New York, NY, USA, 1965. doi:10.1145/1464039.1464045.

[TJS95] M. Tremblay, B. Joy, and K. Shin. A three dimensional register file for superscalar processors. In *Proc. 28th Hawaii International Conference on System Sciences*, volume 1, pages 191–201, January 1995. doi:10.1109/HICSS.1995.375394.

[TLYL04] Zhangxi Tan, Chuang Lin, Hao Yin, and Bo Li. Optimization and benchmark of cryptographic algorithms on network processors. *IEEE Micro*, 24(5):55–69, September/October 2004. ISSN 0272-1732. doi:10.1109/MM.2004.54.

[TP06] Scott E. Thompson and Srivatsan Parthasarathy. Moore's law: the future of Si microelectronics. *Materials Today*, 9(6):20–25, 2006. ISSN 1369-7021. doi:10.1016/S1369-7021(06)71539-5.

[TR86] Andrew S. Tanenbaum and Robbert Van Renesse. Research issues in distributed operating systems. In *Computing in High-Energy Physics*, pages 35–46. North-Holland, 1986.

[TS06] Andrew S. Tanenbaum and Maarten Van Steen. *Distributed Systems: Principles and Paradigms*. Prentice Hall, 2nd edition, June 2006. ISBN 978-0-13-239227-3.

[TVR85] Andrew S. Tanenbaum and Robbert Van Renesse. Distributed operating systems. *ACM Comput. Surv.*, 17:419–470, December 1985. ISSN 0360-0300. doi:10.1145/6041.6074.

[UPC05] UPC Consortium. UPC Language Specifications, v1.2. Technical Report LBNL-59208, Lawrence Berkeley National Lab, May 2005.

[UvTJ11] M. I. Uddin, M. W. van Tol, and C. R. Jesshope. High level simulation of SVP many-core systems. *Parallel Processing Letters*, 21(4):413–438, December 2011. ISSN 0129-6264. doi:10.1142/S0129626411000308.

[van06] Michiel W. van Tol. Exceptions in a microthreaded architecture. Master's thesis, University of Amsterdam, December 2006. Available from: http://dist.svp-home.org/doc/michiel-van-tol-exceptions.ps.

[VD05] Hans Vandierendonck and Koen De Bosschere. XOR-based hash functions. *IEEE Trans. Comput.*, 54(7):800–812, July 2005.

[vEB90] Peter van Emde Boas. Machine models and simulations. In *Handbook of theoretical computer science (vol. A)*, pages 1–66. MIT Press, Cambridge, MA, USA, 1990. ISBN 0-444-88071-2.

[vECGS92] Thorsten von Eicken, David E. Culler, Seth Copen Goldstein, and Klaus Erik Schauser. Active messages: a mechanism for integrated communication and computation. In *ISCA '92: Proc. 19th annual International Symposium on Computer Architecture*, pages 256–266. ACM, New York, NY, USA, 1992. ISBN 0-89791-509-7. doi:10.1145/139669.140382.

[vH07] Eric von Hippel. Horizontal innovation networks–by and for users. *Industrial and Corporate Change*, 16(2):293–315, May 2007. doi:10.1093/icc/dtm005.

[vHvK03] Eric von Hippel and Georg von Krogh. Open source software and the "private-collective" innovation model: Issues for organization science. *Organization Science*, 14(2):209–223, 2003. ISSN 10477039. Available from: http://www.jstor.org/stable/4135161.

[VJ07] Thuy Duong Vu and Chris R. Jesshope. Formalizing SANE virtual processor in thread algebra. In Michael Butler, Michael Hinchey, and María Larrondo-Petrie, editors, *Formal Methods and Software Engineering*, volume 4789 of *Lecture Notes in Computer Science*, pages 345–365. Springer Berlin / Heidelberg, 2007. ISBN 978-3-540-76648-3. doi:10.1007/978-3-540-76650-6_20.

[vTJ11] Michiel W. van Tol and Chris R. Jesshope. An operating system strategy for general-purpose parallel computing on many-core architectures. *Advances in Parallel Computing*, High Performance Computing: From Grids and Clouds to Exascale(20):157–181, 2011. ISBN 978-1-60750-802-1. ISSN 0927-5452. doi:10.3233/978-1-60750-803-8-157.

[vTJLP09] M. W. van Tol, C. R. Jesshope, M. Lankamp, and S. Polstra. An implementation of the SANE Virtual Processor using POSIX threads. *J. Syst. Archit.*, 55(3):162–169, 2009. ISSN 1383-7621.

[vTK11] Michiel W. van Tol and Juha Koivisto. Extending and implementing the self-adaptive virtual processor for distributed memory architectures. Technical Report arXiv:1104.3876v1 [cs.DC], University of Amsterdam and VTT, Finland, 2011. Available from: http://arxiv.org/abs/1104.3876.

[WA09] David Wentzlaff and Anant Agarwal. Factored operating systems (fos): the case for a scalable operating system for multicores. *SIGOPS Oper. Syst. Rev.*, 43:76–85, April 2009. ISSN 0163-5980. doi:10.1145/1531793.1531805.

[WCA02] Emmett Witchel, Josh Cates, and Krste Asanović. Mondrian memory protection. In *Proc. 10th international conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS'02)*, ASPLOS-X, pages 304–316. ACM, New York, NY, USA, 2002. ISBN 1-58113-574-2. doi:10.1145/605397.605429.

[Web00] Steven Weber. The political economy of open source software. BRIE Working Paper 140, University of Berkeley, California, June 2000. Available from: http://brie.berkeley.edu/publications/wp140.pdf.

[Wir95] N. Wirth. A plea for lean software. *Computer*, 28(2):64–68, February 1995. ISSN 0018-9162. doi:10.1109/2.348001.

[WJ96] S.J.E. Wilton and N.P. Jouppi. Cacti: an enhanced cache access and cycle time model. *Solid-State Circuits, IEEE Journal of*, 31(5):677–688, may 1996. ISSN 0018-9200. doi:10.1109/4.509850.

[WM95] Wm. A. Wulf and Sally A. McKee. Hitting the memory wall: implications of the obvious. *SIGARCH Comput. Archit. News*, 23:20–24, March 1995. ISSN 0163-5964. doi:10.1145/216585.216588.

[XMA+10] Guoqing Xu, Nick Mitchell, Matthew Arnold, Atanas Rountev, and Gary Sevitsky. Software bloat analysis: finding, removing, and preventing performance problems in modern large-scale object-oriented applications. In *Proceedings of the FSE/SDP workshop on Future of software engineering research*, FoSER '10, pages 421–426. ACM, New York, NY, USA, 2010. ISBN 978-1-4503-0427-6. doi:10.1145/1882362.1882448.

[YLT05] Yao Yue, Chuang Lin, and Zhangxi Tan. NPCryptBench: a cryptographic benchmark suite for network processors. *SIGARCH Comput. Archit. News*, 34(1):49–56, September 2005. ISSN 0163-5964. doi:10.1145/1147349.1147359.

[YMBYG08] Karim Yaghmour, Jon Masters, Gilad Ben-Yossef, and Philippe Gerum. *Building Embedded Linux Systems*. O'Reilly Media, 2nd edition, 2008. ISBN 0596529686.

[YSP+98] Kathy Yelick, Luigi Semenzato, Geoff Pike, Carleton Miyamoto, Ben Liblit, Arvind Krishnamurthy, Paul Hilfinger, Susan Graham, David Gay, Phil Colella, and Alex Aiken. Titanium: a high-performance Java dialect. *Concurrency: Practice and Experience*, 10(11-13):825–836, November 1998. ISSN 1096-9128. doi:10.1002/(SICI)1096-9128(199809/11)10:11/13<825::AID-CPE383>3.0.CO;2-H.

[ZBMS10] D. Ziakas, A. Baum, R.A. Maddox, and R.J. Safranek. Intel QuickPath Interconnect architectural features supporting scalable system architectures. In *Proc. 18th Annual Symposium on High Performance Interconnects*, HOTI'10, pages 1–6, August 2010. doi:10.1109/HOTI.2010.24.

[ZCHB03] V.V. Zhirnov, III Cavin, R.K., J.A. Hutchby, and G.I. Bourianoff. Limits to binary logic switch scaling - a gedanken model. *Proc. IEEE*, 91(11):1934–1939, nov 2003. ISSN 0018-9219. doi:10.1109/JPROC.2003.818324.

[Zit08] Jonathan Zittrain. *The Future of the Internet—And How to Stop It*. Yale University Press, first edition, April 2008. ISBN 978-0300124873. Available from: http://futureoftheinternet.org/download.

[ZJ07] Li Zhang and Chris R. Jesshope. On-Chip COMA Cache-Coherence Protocol for Microgrids of Microthreaded Cores. In Bouge and et al., editors, *Euro-Par Workshops*, volume 4854 of *LNCS*, pages 38–48. Springer, 2007.

Index