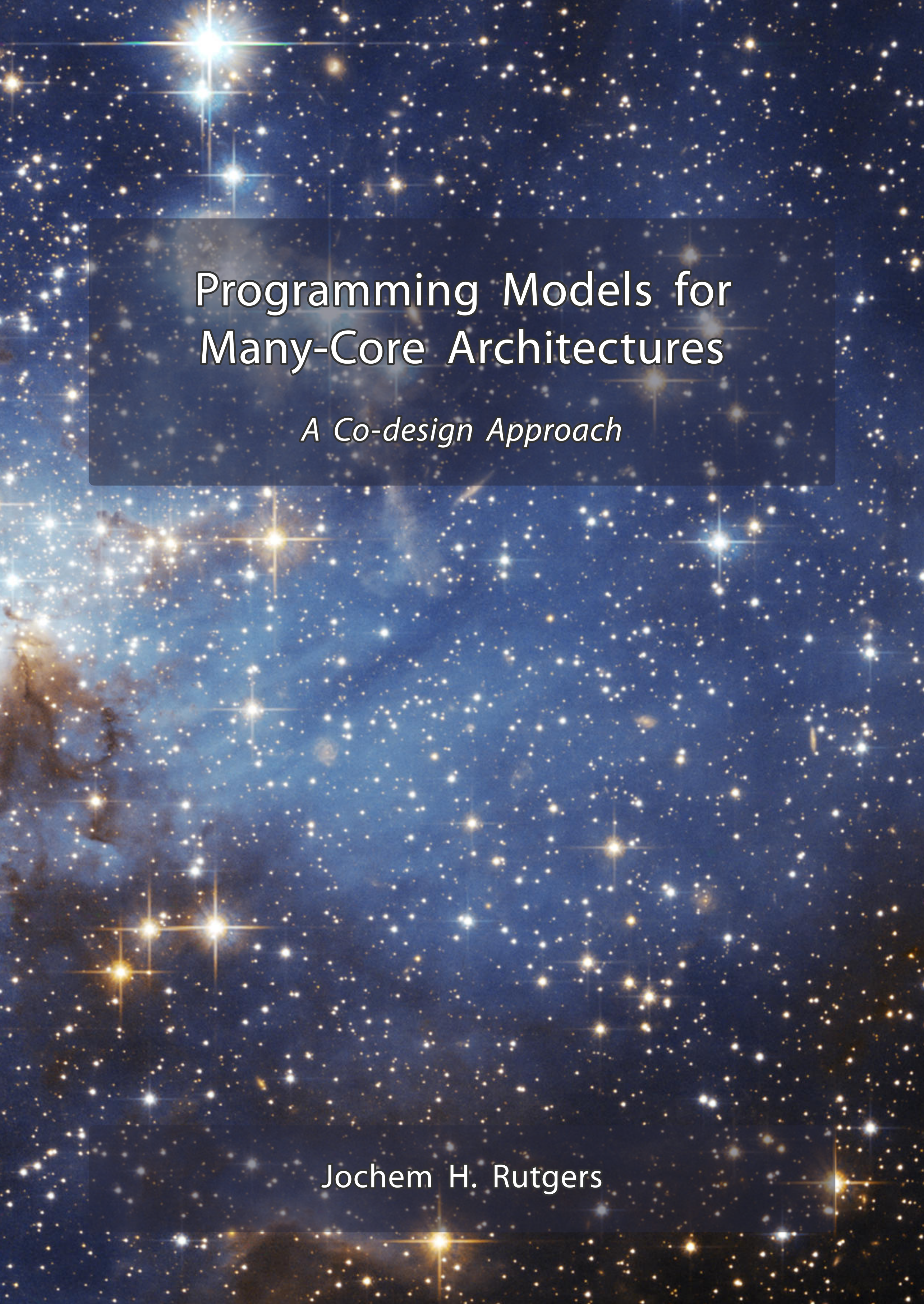


Programming Models for Many-Core Architectures

A Co-design Approach

Jochem H. Rutgers

UNIVERSITY OF TWENTE.

Faculty of Electrical Engineering, Mathematics and Computer Science, Computer Architecture for Embedded Systems (CAES) group

CTIT

CTIT Ph.D. Thesis Series No. 14-292
Centre for Telematics and Information Technology
PO Box 217, 7500 AE Enschede, The Netherlands

STW
Enabling new technology

This research has been conducted within the Netherlands Streaming (NEST) project (project number 10346). This research is supported by the Dutch Technology Foundation STW, which is part of the Netherlands Organisation for Scientific Research (NWO) and partly funded by the Ministry of Economic Affairs.

This thesis was typeset using LaTeX, Ti*k*Z, and Vim. This thesis was printed by Gildeprint Drukkerijen, The Netherlands.

ISBN	978-90-365-3611-0
ISSN	1381-3617; CTIT Ph.D. Thesis Series No. 14-292
DOI	10.3990/1.9789036536110

Programming Models for Many-Core Architectures

A Co-design Approach

Proefschrift

ter verkrijging van
de graad van doctor aan de Universiteit Twente,
op gezag van de rector magnificus,
prof. dr. H. Brinksma,
volgens besluit van het College voor Promoties
in het openbaar te verdedigen
op woensdag 14 mei 2014 om 14.45 uur

door

Jochem Hendrik Rutgers

geboren op 18 april 1984
te Winterswijk

Dit proefschrift is goedgekeurd door:

Prof. dr. ir.	M. J. G. Bekooij	(promotor)
Prof. dr. ir.	G. J. M. Smit	(promotor)

ISBN 978-90-365-3611-0

Abstract

It is unlikely that general-purpose single-core performance will improve much in the coming years. The clock speed is limited by physical constraints, and recent architectural improvements are not as beneficial for performance as those were several years ago. However, the transistor count and density per chip still increase, as feature sizes reduce, and material and processing techniques improve. Given a limited single-core performance, but plenty of transistors, the logical next step is towards *many-core*.

A many-core processor contains at least tens of cores and usually distributed memory, which are connected (but physically separated) by an interconnect that has a communication latency of multiple clock cycles. In contrast to a multicore system, which only has a few tightly coupled cores sharing a single bus and memory, several complex problems arise. Notably, many cores require many parallel tasks to fully utilize the cores, and communication happens in a distributed and decentralized way. Therefore, programming such a processor requires the application to exhibit concurrency. Moreover, a concurrent application has to deal with memory state changes with an observable (non-deterministic) intermediate state, whereas single-core applications observe all state changes to happen atomically. The complexity introduced by these problems makes programming a many-core system with a single-core-based programming approach notoriously hard.

The central concept of this thesis is that abstractions, which are related to (many-core) programming, are structured in a single platform model. A platform is a layered view of the hardware, a memory model, a concurrency model, a model of computation, and compile-time and run-time tooling. Then, a *programming model* is a specific view on this platform, which is used by a programmer.

In this view, some details can be hidden from the programmer's perspective, some details cannot. For example, an operating system presents an infinite number of parallel virtual execution units to the application—details regarding scheduling and context switching of processes on one core are hidden from the programmer. On the other hand, a programmer usually has to take full control over separation, distribution, and balancing of workload among different worker threads. To what extent a programmer can rely on automated control over low-level platform-specific details is part of the programming model. This thesis presents modifications to different abstraction layers of a many-core architecture, in order to make the system as a whole more efficient, and to reduce the complexity that is exposed to the programmer via the programming model.

For evaluation of many-core hardware and corresponding (concurrent) programming techniques, a 32-core MicroBlaze system, named Starburst, is designed and implemented on FPGA. On the hardware architecture level, a network-on-chip is presented that is tailored towards a typical many-core application communication pattern. All cores can access a shared memory, but as this memory becomes a bottleneck, inter-core communication bypasses memory by means of message passing between cores and scratchpad memories. Using message passing and local memories, a new distributed lock algorithm is presented to implement mutexes. The network scales linearly in hardware costs to the number of cores, and the performance of the system scales close to linear (until bounded by memory bandwidth).

Different many-core architectures implement different memory models. However, they have in common that atomicity of state changes should be avoided to reduce hardware complexity. This typically results in a (weak) memory model that does not require caches to be coherent, and processes that disagree on the order of write operations. Moreover, porting applications between hardware with a different memory model requires intrusive modifications, which is error-prone work. In this thesis, a memory model abstraction is defined, which hides the memory model of the hardware from the programmer, and reduces hardware complexity by reducing the atomicity requirements to a minimum, but still allows an efficient implementation for multiple memory hierarchies. Experiments with Starburst demonstrate that software cache coherency can transparently be applied to applications that use this memory model abstraction.

A common approach to exploit the parallel power of a many-core architecture is to use the threaded concurrency model. However, this approach is based on a sequential model of computation, namely a register machine, which does not allow concurrency easily. In order to hide concurrency from the programmer, a change in the model of computation is required. This thesis shows that a programming model based on λ-calculus instead is able to hide the underlying concurrency and memory model. Moreover, the implementation is able to cope with higher interconnect latencies, software cache coherency, and the lack of atomicity of state changes of memory, which is demonstrated using Starburst. Therefore, this approach matches the trends in scalable many-core architectures.

The changes to the abstraction layers and models above have influence on other abstractions in the platform, and especially the programming model. To improve the overall system and its programmability, the changes that seem to improve one layer should fit the properties and goals of other layers. Therefore, this thesis applies *co-design* on all models. Notably, co-design of the memory model, concurrency model, and model of computation is required for a scalable implementation of λ-calculus. Moreover, only the combination of requirements of the many-core hardware from one side and the concurrency model from the other leads to the memory model abstraction above. Hence, this thesis shows that to cope with the current trends in many-core architectures from a programming perspective, it is essential and feasible to inspect and adapt all abstractions collectively.

Samenvatting

Het is onwaarschijnlijk dat de komende tijd de rekenkracht van een single-coreprocessor veel zal verbeteren. De kloksnelheid is beperkt door natuurkundige limieten en recente verbeteringen aan het ontwerp geven niet een snelheidswinst zoals die van enkele jaren geleden gaven. Toch nemen het aantal transistors per chip en de dichtheid nog steeds toe, omdat de gebruikte materialen en de productietechnieken blijven verbeteren. De combinatie van de beperkte rekenkracht van een enkele core en een overvloed aan transistors zal logischerwijs leiden tot *many-core*-processoren.

Een many-coreprocessor bevat minstens tientallen cores en meestal gedistribueerd geheugen, die zijn verbonden (maar fysiek gescheiden) door een netwerk waarin communicatie meerdere klokcycli in beslag neemt. Ten opzichte van een multicoreprocessor, waarin slechts enkele met elkaar verweven cores zitten en één bus en geheugen delen, worden een aantal complexe problemen zichtbaar. Het hebben van veel cores vraagt om veel parallelle taken om alle cores te benutten. Daarbij is communicatie gedistribueerd en decentraal geregeld. Om een dergelijke processor te kunnen programmeren moet de applicatie daarom ontworpen zijn voor parallellisme. Daarnaast moet deze applicatie kunnen omgaan met toestandsveranderingen van geheugen, waarbij de toestandsovergang non-deterministisch is, in tegenstelling tot sequentiële applicaties waarvoor toestandsveranderingen atomair lijken. De complexiteit als gevolg van deze problemen maakt het programmeren van een many-coresysteem met single-coreprogrammeertechnieken zeer moeilijk.

Het centrale concept van dit proefschrift is dat abstracties die gerelateerd zijn aan (parallel) programmeren, zijn gestructureerd in één platformmodel. Een platform is een gelaagde weergave van de hardware, het geheugenmodel, concurrencymodel, berekeningsmodel en de software voor compilatie en executie. Het *programmeermodel* is een specifiek perspectief op dit platform voor de programmeur.

Dit perspectief kan bepaalde details voor de programmeur verbergen of benadrukken. Een besturingssysteem biedt bijvoorbeeld een oneindig aantal virtuele processoren aan een applicatie—hoe de rekentijd van een processor wordt verdeeld over de processen wordt verborgen voor de programmeur. Echter, een programmeur wordt wel geacht exact aan te geven hoe rekenwerk moet worden opgesplitst en verdeeld over verschillende processen. Het programmeermodel geeft aan in hoeverre een programmeur kan vertrouwen op correcte aansturing van platformspecifieke details. Dit proefschrift beschrijft aanpassingen aan de verschillende abstractielagen, die het systeem als geheel efficiënter maken en de complexiteit reduceren waar de programmeur via het programmeermodel aan wordt blootgesteld.

Voor evaluatie van parallelle hardware en bijbehorende programmeertechnieken is er een 32-core MicroBlaze-systeem voor FPGA ontwikkeld, genaamd Starburst. Het bevat een netwerk dat is toegespitst op gangbare communicatiepatronen van many-coreapplicaties. De cores delen een geheugen. Echter, cores kunnen dit geheugen omzeilen door berichten uit te wisselen via kleine, lokale geheugens als bandbreedte naar het gedeelde geheugen een knelpunt wordt. Op basis van deze berichten tussen cores, is een gedistribueerd mutex-algoritme ontworpen. De hardwarekosten van het netwerk schalen lineair mee met het aantal cores. De totale rekenkracht van het systeem schaalt bijna lineair (totdat de geheugenbandbreedte verzadigd raakt).

Verschillende many-corearchitecturen ondersteunen verschillende geheugenmodellen. Deze hebben als overeenkomst dat atomaire toestandsveranderingen vermeden worden om de hardware eenvoudiger te maken. Het resulterende (zwakke) geheugenmodel vereist doorgaans niet dat caches coherent zijn, noch dat alle processen schrijfoperaties naar geheugen in dezelfde volgorde zien. Daarnaast vraagt het omschrijven van applicaties voor hardware met een ander geheugenmodel om ingrijpende aanpassingen. Dit is foutgevoelig werk. In dit proefschrift wordt een geheugenmodelabstractie gedefinieerd. Deze verbergt het geheugenmodel van de hardware voor de programmeur en versimpelt de hardware-implementatie, doordat de eisen aan de atomiciteit van toestandsveranderingen zijn versoepeld. Toch kan de abstractie efficiënt worden geïmplementeerd op verschillende geheugenarchitecturen. Experimenten met Starburst laten zien dat software cache coherency automatisch kan worden toegepast op applicaties die deze abstractie gebruiken.

Doorgaans wordt het threadingmodel gebruikt om parallellisme van de hardware te benutten. Echter, dit model is gebaseerd op een sequentieel berekeningsmodel, namelijk een registermachine, die concurrency niet eenvoudig toelaat. Een ander berekeningsmodel is nodig om concurrency voor de programmeur te verbergen. Dit proefschrift laat zien dat een op λ-calculus gebaseerd programmeermodel het onderliggende concurrency- en geheugenmodel wel kan verbergen. Tevens kan de implementatie voor Starburst omgaan met trage netwerkcommunicatie, software cache coherency en niet-atomaire toestandsveranderingen. Deze aanpak past daarom goed bij de trends in schaalbare many-corearchitecturen.

Aanpassingen in de abstractielagen en bovengenoemde modellen hebben invloed op andere abstracties in het platform, maar voornamelijk op het programmeermodel. Om het systeem als geheel en de programmeerbaarheid te verbeteren moeten verbeteringen in de ene abstractielaag passen bij de eigenschappen van andere lagen. Daarom wordt in dit proefschrift *co-design* toegepast op alle modellen. Co-design van het geheugenmodel, concurrencymodel en berekeningsmodel is bijvoorbeeld noodzakelijk voor een schaalbare implementatie van λ-calculus. Daarnaast leidt alleen de combinatie van eisen van many-corehardware van de ene kant en het concurrencymodel van de andere kant tot de genoemde geheugenmodelabstractie. Dit proefschrift laat dus zien dat het essentieel en haalbaar is om alle abstracties gezamenlijk te beschouwen en aan te passen, om vanuit een programmeerperspectief om te kunnen gaan met de huidige trends in many-corearchitecturen.

Dankwoord

Eén woord per acht minuten. Als je dat gedurende het promotietraject toevoegt aan je proefschrift, dan heb je aan het eind voldoende omvangrijk werk geleverd. Gegeven een gemiddelde typesnelheid van 50 woorden per minuut, ben je dus 0,25 % van je tijd bezig met je proefschrift. Dat geeft lekker veel tijd voor het onderzoek zelf.

Ik weet nog goed dat ik tegen het eind van mijn afstuderen uit het raam staarde en dacht: "Het zit er bijna op, hier op de universiteit …" Het treurige gevoel dat opkwam, gaf aan dat de omgeving, het onderwerp en het onderzoek leuk genoeg zijn om er nog een tijdje aan vast te knopen. Ik vind het leuk om met de techniek bezig te zijn en als aio krijg je ruim de tijd om ergens lekker diep in te duiken. En het schrijven van papers en een proefschrift? Ach, het draait toch om de inhoud, dus dat zal wel loslopen … Ik ben mijn opa en Jan dankbaar voor de stimulans om eraan te beginnen. Daarnaast heeft Gerard een goede plek geboden, zodat ik kon uitpluizen wat ik leuk vind.

Zoals iedere aio wel weet, kost schrijven toch iets meer tijd dan die 0,25 %. Echter, de scheidslijn tussen onderzoek doen en schrijven is heel dun; door de ideeën op papier uit te werken, krijgen ze vorm en worden ze scherper. En Marco, die al vroeg in het traject als begeleider betrokken raakte, heeft hier een zeer waardevolle bijdrage aan geleverd met zijn kennis, kritiek en uitdagingen.

Hoewel uiteindelijk alleen mijn naam op dit proefschrift staat, zijn er veel mensen die direct of indirect invloed hebben gehad op dit resultaat. In willekeurige volgorde bedank ik enkelen van deze: mijn kamergenoten Marco, Arjan, Robert en Koen voor de nodige discussies, overdenkingen en afleiding tijdens die stressvolle proefschriftschrijfdagen; Berend die zich moedig durfde te wagen in de donkere hoekjes[1] van Starburst; Bert, die het weer moest ontgelden wanneer er een (NFS-)server haperde, ook al kon hij daar niets aan doen; Marlous, Thelma en Nicole voor de ondersteuning bij allerlei praktische zaken, zoals het boeken van snoepreisjes naar conferenties; Pascal en Philip voor de geboden doorgroeimogelijkheden en een basis van een LaTeX-template voor dit proefschrift, die via een ingewikkelde reeks van afgeleiden door (onder anderen?) Albert, Vincent, Maurice en Timon bij mij is gekomen, waar ook ik vervolgens hier en daar een packagetegenaan heb geknutseld; Hermen, die het is gelukd om de allerlaatste typefout uit m'n proefschrift te halen; Christiaan, Mark en alle anderen van CAES voor de pauzes en borrels, die altijd weer met een hoop lol en onzin werden gevuld.

[1] Ho eens even, zoveel van die hoekjes zijn er helemaal niet!

De vier jaar (plus een beetje) zijn omgevlogen. Ik heb genoten van wat ik heb gezien, geleerd en gedaan. Mijn ouders hebben mij altijd gesteund en hebben meegeleefd tijdens verre reizen, waar ik ze erg dankbaar voor ben, maar ze kunnen het toch maar lastig volgen wat ik nu precies allemaal heb gedaan, ondanks mijn verwoede pogingen om het uit te leggen. Wellicht dat dit proefschrift helderheid biedt, want eindelijk staat het nu eens allemaal netjes bij elkaar …

Voor de laatste loodjes krijg ik hulp van mijn paranimfen, mijn beste vriend en geregelde lunchwandelgenoot Martijn, en aanstaande schoonvader Henk. Tot slot, ik ben heel blij met mijn geliefde, Marjan, die het altijd weer lukt om me op te vrolijken, wanneer er bijvoorbeeld een paper werd afgewezen, en die ik vaker dan eens heb beloofd om nu eens eerder thuis te komen, zodat we om een fatsoenlijke tijd kunnen eten.

Jochem
Enschede, april 2014

Contents

1 **Introduction** 1

1.1 Multicore and many-core . . . 3

1.2 Abstraction . . . 4

1.3 Embedded systems . . . 6

1.4 Problem statement and approach . . . 6

1.5 Contributions . . . 7

1.6 Structure . . . 8

2 **Trends in Many-Core Architectures** 11

2.1 Ten many-core architectures . . . 11

2.2 Simpler cores, more performance . . . 12

2.3 Transparent processing tiles . . . 14

2.4 Interconnect: coherency traffic vs. data packets . . . 15

2.5 Weak-memory hierarchy . . . 16

2.6 Programming model consensus . . . 18

2.7 Starburst . . . 19

2.7.1 System overview . . . 19

2.7.2 OS: Helix . . . 20

2.7.3 Application environment . . . 22

2.8 Benchmark applications . . . 22

2.8.1 SPLASH-2 . . . 23

2.8.2 PARSEC . . . 24

2.8.3 NoFib . . . 25

3 Platforms and Programming Abstraction 27

3.1 Many-core hardware is the driving force . . . 28

3.2 Memory model—the hardware's contract . . . 29

3.3 A concurrency model to orchestrate interaction . . . 31

3.4 Computation and algorithms . . . 33

3.5 Programming model: a peephole view . . . 35

3.5.1 Existing programming language's models . . . 35

3.5.2 Less is more . . . 38

3.6 Platform and portability . . . 39

3.7 Related work . . . 41

3.8 Conclusion . . . 42

4 Efficient Hardware Infrastructure 45

4.1 Communication patterns and topology . . . 47

4.2 Baseline: Starburst with Æthereal . . . 48

4.2.1 8-core setup . . . 48

4.2.2 Synthesis results: exponential costs . . . 50

4.2.3 Core utilization by benchmark applications . . . 51

4.2.4 Shortcomings of connection orientation . . . 52

4.3 Warpfield: a connectionless NoC . . . 53

4.3.1 Bitopological architecture . . . 53

4.3.2 Improvements in hardware and software . . . 56

4.3.3 Bounded temporal behavior . . . 57

4.4 Inter-core synchronization profile . . . 60

4.4.1 Polling main memory measurements . . . 61

4.4.2 Mutex locality . . . 63

4.5 Asymmetric distributed lock algorithm . . . 64

4.5.1 Existing synchronization solutions . . . 64

4.5.2 The algorithm: a three-party asymmetry . . . 65

4.5.3 Experimental comparison results . . . 68

4.5.4 Locality trade-off . . . 70

4.6 Hardware and performance scalability . . . 72

4.7 Conclusion . . . 74

5 Usable Weak Memory Model 77
5.1 The problem with memories 78
5.1.1 *Various memory models* 79
5.1.2 *PMC's basic idea* 84
5.2 A Portable Memory Consistency model 86
5.2.1 *Fundamentals* 86
5.2.2 *Operations by processes* 87
5.2.3 *Orderings: semantics of operations* 88
5.2.4 *Observing slowly* 91
5.2.5 *Comparison to existing models* 92
5.3 Annotation and abstraction 93
5.3.1 *Front-end: annotations in source code* 93
5.3.2 *Back-end example: three views on Starburst* 95
5.4 Case study 100
5.4.1 *Software cache coherency: SPLASH-2 benchmark* 100
5.4.2 *Distributed shared memory: multi-reader/-writer FIFO* 102
5.4.3 *Scratchpad memory: motion estimation* 103
5.5 Conclusion 105

6 Implicit Software Concurrency 109
6.1 Basic idea 111
6.2 Related work 112
6.3 Shift in paradigm: λ-calculus and its implementation 113
6.3.1 *Background on λ-calculus* 113
6.3.2 *Our simple functional language: LambdaC++* 115
6.3.3 *The atomic-free core: data races and lossy work queue* 124
6.4 Impact on memory consistency and synchronization 126
6.4.1 *A λ-term's life and rules* 126
6.4.2 *Mapping from rules to PMC* 128
6.5 Experiments 132
6.5.1 *Scalability and speedup* 132
6.5.2 *Locality and overhead* 135
6.6 Conclusion 136

7 Conclusion 139
7.1 Contributions 142
7.2 Recommendations for future work 143

A Etymology 147
A.1 Starburst . . . 147
A.2 Warpfield . . . 147
A.3 Helix . . . 148
A.4 skat . . . 148
A.5 LambdaC++ . . . 149

Acronyms 153

Bibliography 157

List of Publications 169

Index 171

1 Introduction

Abstract – *Processors incorporate more and more cores. With the increasing core count, it becomes harder to implement convenient features like atomic operations, ordering of all memory operations, and hardware cache coherency. When these features are not supported by the hardware, applications become more complex. This makes programming these many-core architectures hard. This thesis defines programming models for many-core architectures, such that current trends in processor design can be dealt with. Finding a good balance between choices regarding different layers of the platform is essential in order to ease programming. Throughout the thesis, design choices and consequences are evaluated based on a co-design of hardware and software abstraction layers.*

"The single-core era has ended, multicore processors are here to stay. Getting 'free' computing power by just increasing the clock frequency, does not work anymore. So, when one processor does not get faster, just use multiple of them." This has been said many times, and it is illustrated in figure 1.1 on the following page. In 2005, both Intel and AMD introduced a multicore processor, which marks an important transition. In the past, parallelism was only achieved by putting multiple processors together for specific systems like servers and supercomputers. Now, processors make every (consumer) system a parallel machine. Programmers accept the fact that they have to face programming for concurrency. Although it might sound like a reasonable conclusion, 'just' having multiple cores only adds *raw* computing power, but does not imply that software can make use of it.

Software for a single-core system behaves in a way programmers can understand easily. Instructions are executed in the order that is defined, and if designed carefully, the program always gives a correct result. When the computer becomes faster, e.g., runs at a higher clock speed, the software will run faster. Many (single-core) technological enhancements, like smaller feature sizes, caches, and branch prediction, can be applied to the processor architecture and improve performance without changes to the software.

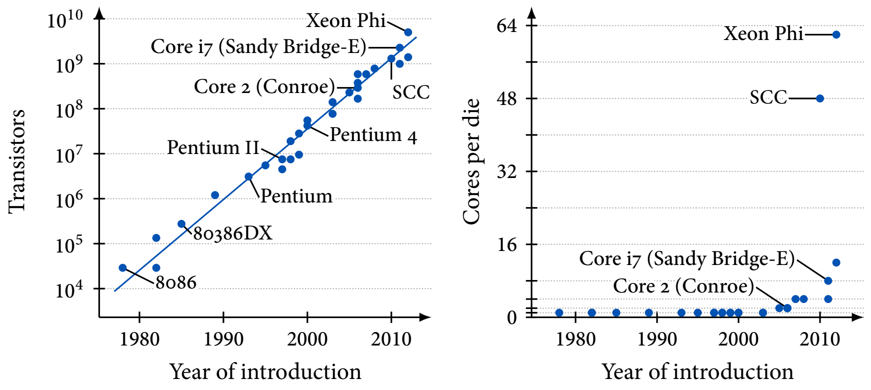


Figure 1.1 – Various Intel microprocessors

In great contrast, exploiting hardware parallelism (in the form of multiple cores) by concurrent programming can only be accomplished when the software is changed. The program has to be split in chunks of work that can be executed in parallel. Based on the single-core programming principles, programming involves defining a somewhat balanced set of communicating instruction sequences. When a multicore computer becomes faster—which effectively means that more cores are added—a properly balanced multi-threaded program might not even benefit at all. More importantly, when the speed of a hardware component changes, the latency and interleaving of communication might change too and even break the program. Such bugs are hard to find, even harder to reproduce, and therefore almost impossible to fix properly. Moving from single-core to multicore is one example of an improvement to a computer system as a whole, which requires changes to multiple aspects of the design of such a system; this truly requires *co-design* of the hardware architecture and the programming approach.

Hardware and software have influenced each other for a long time. An example of the hardware–software interplay is the addition of threading to the latest C and C++ standards, as a response to multicore hardware. The introduction of vector instructions in general-purpose processors as a response to the increasing demand for graphics processing, is just another example. So, co-design is commonly applied in processor design.

Nevertheless, several trends are visible that pushes hardware complexity via the programming model to the application. Borkar and Chien [22] conclude that the performance of hardware can only be increased under acceptable energy demands, when software supports these changes to the hardware. While hardware can only respond to events that occur at this moment, software has some knowledge and control over the future, e.g., by scheduling. Therefore, software might be able to reduce the power usage more than hardware can do, by taking control over fine-grained dynamic power management, such as turning off cores that are not to be used soon. A different trend shows that the performance of memory technology scales not as fast as the performance of logic circuitry, so memory becomes a bottleneck, and the software should exploit data locality even more. Therefore, the memory hierarchy becomes more complex, e.g., because of multiple levels of caches, and control over this hierarchy lies in the hands of software [29]. Another trend can be seen in how concurrency is handled. More parallelism can only be realized by a change in the programming paradigm [105]. However, it is hard to accomplish this. Threading is a popular approach, but it introduces non-determinism at such a scale, that it is hard to oversee and control by a programmer [71]. Additionally, threading libraries might break optimizations by a concurrency-agnostic compiler [20]. Among many other APIs, OpenMP [36] allows fine-grained control over parallelism, which a compiler cannot statically determine by itself, by means of annotations in the C source code. In all these trends, handling of low-level machine-specific details is based on analysis of, or control by, the high-level (pseudo–)machine-*independent* application. In practice, however, the programmer has to do it by hand…

It is logical to expose new hardware features to the programmer first, and rely on manual control; it takes time until the feature is understood well enough to take care of it automatically. However, the ultimate goal is to let a tool do all the work that can be done automatically. In case of the aforementioned multicore trends, parallelism and the memory hierarchy are features that are hard to handle correctly by hand. The question is whether it can be automated or not, and what the consequences are. This thesis will discuss consequences of choices regarding various abstraction layers that are relevant for programming a multicore system.

1.1 Multicore and many-core

Let us first define such a 'multicore' system in the context of this thesis. A parallel machine can be organized in many ways, such as: multiple cores within a processor, communicating via an on-chip bus; multiple processors within a computer, communicating via an off-chip bus; and multiple computers within a cluster, communicating via Ethernet. These architectures all have their benefits and drawbacks.

One interesting property is the latency of communication. As an example, different latencies of reads within the Intel Nehalem processor are listed in table 1.1 on the next page [81]. It shows that off-chip communication takes a considerable amount of time, compared to reads from memories that are closer. Combined with the trend of figure 1.1, the continuing exponential growth of the number of transistors per chip gives resources and a performance benefit to integrate more cores on one die. Hence, it is likely that the number of cores per processor will grow exponentially.

Multicore systems are often classified as *many-core* to express a high core count. It also informally tends to stress the need for specific techniques that are related to

Table 1.1 – Read latency (Intel Nehalem) [81]

data source	latency (cycles)
local L1 cache	4
local L2 cache	10
local L3 cache	38
other core's cache (same die)	38–83
other core's cache (other die)	102–170
off-chip RAM (same die)	191
off-chip RAM (other die)	310

concurrency. However, the exact difference between multi and many is usually not clearly defined. We use the following definition:

multicore
: *A symmetric multiprocessing* (SMP) *architecture containing tightly coupled identical superscalar cores, under control of a single OS. The cores are tightly coupled in the sense that they (usually) share all memory, and the caches are hardware cache coherent.*

many-core
: *A processor architecture that contains at least tens of loosely coupled (possibly heterogeneous) simpler cores. The cores are loosely coupled in the sense that the memory is characterized as a non-uniform memory architecture* (NUMA), *they (usually) have incoherent caches, and every core runs its own (instance of an) OS.*

Most commercially available processors can be described as multicores. The Intel SCC and Intel Xeon Phi can be classified as many-cores, even though the latter has hardware cache coherency. As the core count increases, hardware cache coherency is unlikely to sustain [29], which will probably make most future processors many-cores.

This thesis focuses on programming a single many-core processor. From a software perspective, this conceptually does not differ much from a multiprocessor setup. Therefore, we use the terms 'core' and 'processor' as synonyms, despite their physical differences.

1.2 Abstraction

Making abstractions, as we just did, is very important in programming. A computer consists of many abstraction layers, which hide details about the implementation. Table 1.2 lists several layers of abstraction within a processor. All these layers are replaceable, without having to redefine all other layers, except for the surrounding onces. For example, when CMOS technology is replaced, the standard cells have to be redesigned, but the processor architecture is (largely) independent of it.

Table 1.2 – Abstraction layers

model	examples
programming language	C, Haskell
…	
logical processors	context switching by OS
instruction set	x86-64, Thumb-2
processor	Phenom II, MicroBlaze
core	IA-32, hyperthreading
components	RAM, ALU
standard cells	and-gate, flip-flop
circuit logic	CMOS
semiconductor	GaAs
atoms	Si, O
Standard Model of particle physics	up quark, muon neutrino

An abstraction generalizes the implementation of it. As such, conclusions drawn, based on the abstraction, should be valid for every implementation. In the examples of table 1.2, two different types of abstraction can be observed: the abstraction contains either *fewer* or *more* details than its implementation. The abstraction of CMOS technology to standard cells hides all details about feature size and thickness of the metal layers. Such an abstraction layer has to fill in the missing details, which usually comes at a cost or overhead—rectangular shaped standard cells do not necessarily use the least amount of chip area. In contrast, the x86-64 is a CISC instruction set, where processors translate it to a RISC set of simpler micro-operations. So, CISC instructions carry more information than what is required by the processor; the implementation of the abstraction layer can make optimal choices, based on the abundant information.

The programming language at the top of the list does not fit in this definition of an abstraction. It partly hides details, such as details of the assembly language of the specific target processor. However, it exposes issues like concurrency and inter-thread communication. For example in C, concurrency is something that has to be done by the programmer. Most importantly, different programming languages hide and expose different aspects. For portability reasons, a proper programming model is required.

The (software-related) abstraction layers at the top of table 1.2 are not as clearly delimited as those at the bottom. The question is whether proper layers can be defined, and in what way implementation details can be hidden from the programmer. In this thesis, we discuss the abstraction layers that are relevant from a programming perspective, and how these abstractions influence each other. A good programming model is designed such that it allows utilizing the raw computing power of a many-core architecture, with a high level of abstraction.

1.3 Embedded systems

If utilizing all computer power, i.e. performance, is not relevant, programming a multicore system is rather straightforward; use one core, and leave all the others idle. For a desktop PC, this might be acceptable in some cases. However, within the embedded domain, resources are more precious or performance requirements stricter. Embedded system processors follow the same trends as figure 1.1 on page 2 illustrates for desktop and server systems. The same technology is used, but there is a time offset of several years in which the technology becomes mature, more energy efficient, and feasible to be used in battery-powered devices. For example, where the first iPhone in 2007 uses a single-core ARM processor, the first quad-core smartphones appeared in 2012.

Moreover, embedded systems are often used in a context where time-critical interaction with the environment is required. Examples include video decoding with a constant frame rate, and control of a car or airplane. In this sense, embedded systems are pushed to their limits, which makes investments in new techniques worthwhile, which in turn can also be applied in general-purpose computing at a later stage. The combination of limited resources and performance requirements in an embedded system makes the multicore programming challenge even more interesting. The techniques presented in this thesis are therefore tailored towards embedded systems, but might also be applicable in other systems.

1.4 Problem statement and approach

As discussed above, processors become increasingly parallel. In an embedded context, it is important to maximize the performance and therefore to utilize all available cores. However, in many-core architectures, concessions to programmability are made by changing several aspects of the architecture in favor of hardware scalability, production costs, reduced design complexity, or energy efficiency. These changes to the hardware are reflected in the programming model, and are currently exposed to the programmer. The central problem this thesis addresses is:

> *How can we cope with the hardware trends in embedded many-core architectures, from a programming perspective?*

The approach is to define *programming models* in a way that the complexity of the trends mentioned above is hidden from the programmer. Then, the compiler and a run-time system should have all information to handle low-level details automatically, efficiently, and correctly. We limit ourselves to the following aspects.

At the hardware-architectural level, a network-on-chip *(NoC)* with a mesh topology is often advocated as a scalable interconnection infrastructure. However, such an interconnect requires routing through the mesh. To guarantee bandwidth between two cores or to memory, the communication pattern of the application is required to determine the allocation of buffers and network links. Such a communication pattern is assumed to be pseudo-static—it is static during a specific phase of the program, but changes over the phases. However, the preferred programming approach, C and threads, does not match the requirement that the communication pattern has to be known on beforehand. Additionally, as the latency in number of clock cycles increases at an increasing core count, atomic operations like a compare-and-swap are hard to realize. When these operations are absent or more expensive, it influences the choices a programmer might make about concurrency. We investigate the *interconnect*, guarantees about inter-core communication, and *synchronization* protocols, and we propose a new interconnect that better suits the needs.

As a NUMA architecture is used, e.g., by using scratchpad memories, the performance is influenced by the location where application data is stored. Moreover, as processors and memories are distributed, a total ordering of operations on them cannot be guaranteed. This makes it harder to reason about the behavior of the memory and therefore the system state. Additionally, hardware cache coherency becomes notoriously complex at high core counts, but incoherent caches make the memory behavior even harder to understand. We define a *memory model* that is able to hide handling caches and scratchpad memories, and allows easily porting applications to other memory architectures.

The actual number of cores is usually not known at compile time. Therefore, the application has to be suitable to be run on any number of cores. This has a major impact on how an application should be designed and written. Defining *concurrency* in an application by hand is error-prone. We discuss a scalable programming approach to do this automatically.

Every layer of abstraction has influence on the surrounding layers. More importantly, choices regarding a lower level have an impact on programming. Therefore, we evaluate all decisions in a *co-design* approach, such that the overall programming efficiency is improved.

1.5 Contributions

The central concept of this thesis is the definition of a programming model with respect to the hardware–software platform. A platform contains a hardware architecture, and implements a memory model and a concurrency model. On top of that, a model of computation is combined with a programming paradigm in a programming model. The programming model exposes specific details of the underlying models, but hides others. We define a layered overview of a programming model, which allows characterization of programming languages, based on what information a programming should give to guide a proper implementation.

We propose a tree-shaped interconnect with a ring for core-to-memory and core-to-core communication, respectively. The interconnect uses a work-conserving distributed first-come-first-served arbitration scheme, and gives bandwidth guarantees. The hardware costs are low and scale linearly to the number of cores.

As atomic read–modify–write operations are hard to realize in a many-core architecture, synchronization is usually based on polling background memory, which is expensive in terms of bandwidth. We present a distributed lock algorithm, which benefits from the local memories and bypasses the background memory.

Cache coherency, scratchpad memories, and distributed shared memory are generalized in our proposed Portable Memory Consistency *(PMC)* model. This model allows abstracting from any memory architecture, while retaining the essential memory operation orderings that are required for programming. We show an implementation to several memory architectures, including software cache coherency, which is transparently applied to standard benchmark applications.

Finally, we present an implementation of a functional language, which utilizes the full parallel capacity of a many-core system. Most interestingly, the implementation is *atomic-free*; no locks, atomic read–modify–write operations, or strong memory model is required. This property allows a further increase of the number of cores, while locality can be exploited transparently.

All experiments are conducted on Starburst, our many-core system on FPGA, using standard benchmark applications. The system reflects the current trends in many-core architectures. This allows evaluation of all aforementioned aspects in a realistic environment.

1.6 Structure

This thesis is organized as depicted by the figure on page 10. The layered view of a many-core platform with the programming model form the core of the thesis.

Chapter 2 discusses trends in many-core architectures and applications. Based on the observed trends, we designed and implemented our experimental platform Starburst. The parallel benchmarks applications from the SPLASH-2 and NoFib are discussed, which we use throughout the thesis for evaluation.

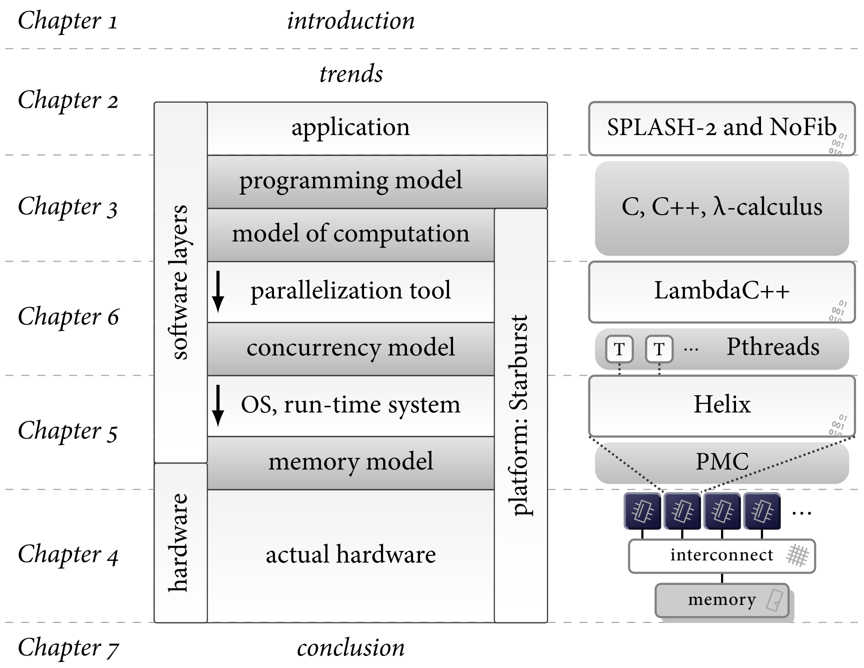
In chapter 3, all layers in the figure are discussed in more detail. The programming model is defined, in terms of the underlying models. To make programming easier, the programming model should hide as many details from the other layers as possible. To this extent, specific optimizations in the remaining layers are discussed in chapters 4 to 6 in a bottom-up fashion.

Chapter 4 discusses the communication infrastructure and synchronization. The tree-shaped interconnect is presented, in combination with core-to-core communication that is required for the distributed lock algorithm. Chapter 5 presents the PMC model and an approach to annotate existing applications in order to be portable to any memory architecture. Chapter 6 presents a method that hides concurrency from the programmer, but still allows utilizing all cores.

Finally, chapter 7 concludes the thesis, formulates the contributions in more detail, and presents recommendations for future work.

This empty page leaves some room for random thoughts:

The universe is inherently parallel, and laws of nature are applied everywhere without computational effort and error margin. Why is it that hard for a computer in the same universe to do a universe-compatible N-body simulation?



Thesis overview

2 Trends in Many-Core Architectures

Abstract – *Based on a comparison of ten contemporary commercial many-core architectures, several trends can be observed. The cores are relatively simple, and memory bandwidth per core is limited. Most architectures have multi-level caches, which are hardware cache coherent. However, weak memory models are used. In contrast to the attention in research, only a few architectures have scratchpad memories. Our many-core architecture, Starburst, captures both commercial and research trends.*

Chapter 1 discussed the trends of microprocessors, and concluded that every processor will become a multicore one. The tendency is that locality is crucial for performance. In this chapter, we discuss several commercial processors in more detail, and relate these to the high-level trends above. For evaluation purposes throughout the thesis, we designed and built the many-core system Starburst, which reflects these trends in current and expected future architectures.

2.1 Ten many-core architectures

Table 2.1 on the following page lists several multicore architectures of the last several years. These architectures all are single chip packages, deliver a high performance by utilizing multiple cores, and are commercially available, except for the experimental Intel SCC. All architectures are targeting high-performance computing, except for the Adapteva Epiphany-IV and Samsung Exynos 4 Quad. These two systems are designed for embedded systems with a limited power budget, like smartphones and tablets.

The table shows the number of cores and the total number of hardware-supported threads. All these cores are homogeneous. Although some systems include several

Table 2.1 – Architectures

	[ref.]	year	core type	cores	(threads)	clock (MHz)
Tilera TILE-Gx8072	[107]	2009	DSP	72		1 000
IBM POWER7	[66]	2010	C1	8	(32)	3 550
Oracle UltraSPARC T3	[96]	2010	SPARC V9	16	(128)	1 649
Intel SCC	[58]	2010	Pentium-100	48		1 000
Intel i7-3930K	[60]	2011	Sandy Bridge-E	6	(12)	3 200
Cavium OCTEON II CN6880	[27]	2011	cnMIPS64 v2	32		1 500
Adapteva Epiphany-IV	[3]	2011	RISC	64		800
Samsung Exynos 4 Quad	[118]	2012	ARM Cortex-A9	4		1 400
Freescale T4240	[42]	2012	Power e6500	12	(24)	1 800
Intel Xeon Phi	[61]	2012	x86-64, vector	61	(244)	1 238
Starburst			MicroBlaze	32		100

accelerators, these are not taken into account for the core count. As the number of transistors per chip increases, it is expected [22] that processors will integrate more heterogeneous cores or more accelerators, but this is not reflected in most of the systems listed—only the Cavium OCTEON II CN6880, Freescale T4240, and Exynos 4 Quad integrate accelerators for graphics or other applications. Starburst[1] (in the form it is discussed in this thesis) is a homogeneous MicroBlaze system with a configurable core count of up to 32. In contrast to all other architectures, it is mapped onto an FPGA, which limits the clock frequency to 100 MHz.

It is clear that contemporary high-end systems already require tens to hundreds of concurrent threads to utilize the full hardware of a single processor. We do not consider (general-purpose) GPUs at this point. Although these processors have thousands of cores, there usability is limited to parallel vector operations, like graphics and specific scientific workloads. Moreover, there are constraints in memory accesses, code and data size, etc. In general, all systems of table 2.1 are (supposed to be) programmed in C with threads, which cannot be done on a GPU.

The setup of these systems is very common: cores have a small L1 instruction and data cache. Often, cores are grouped in clusters of two or four cores, which connect via a low-latency interconnect to a shared L2 cache. Furthermore, the individual cores or clusters are connected via a NoC with a mesh topology or a multilayer bus to each other, and via one or more memory controllers to external DDR memory. The individual properties are discussed in more detail in the subsequent sections.

2.2 Simpler cores, more performance

Single-core processors use the increasing amount of transistors to implement complex microarchitectural features like out-of-order execution, exploiting instruction-

[1]Refer to appendix A for a description where the name comes from.

Table 2.2 – Core and memory performance

	CoreMark score [34]	CoreMark per MHz	shared-memory bandwidth[a] in total (GB/s)	per core (B/cycle)	per CoreMark (MB/s)
Tilera TILE-Gx8072	230 196	230.19	53.6	0.80	0.238
IBM POWER7	336 196	94.70	95.4	3.61	0.291
UltraSPARC T3	87 054	52.79	23.8	1.94	0.280
Intel SCC	102 240	102.24	21	0.44	0.210
Intel i7-3930K	150 962	41.17	47.7	2.67	0.323
OCTEON II CN6880	153 477	102.32	46.6	1.04	0.311
Epiphany-IV	78 749	98.44	6.4	0.13	0.083
Exynos 4 Quad	22 243	15.89	6.4	1.23	0.295
Freescale T4240	187 874	104.37	46.9	2.33	0.256
Starburst	4 521	45.21	0.391	0.13	0.088

[a] Peak core–shared-memory bandwidth, based on the memory controller or the interface to the interconnect

level parallelism *(ILP)*, and deep pipelines. These features are relatively costly, compared to the gained speedup [22]. Once the burden to multicore and concurrency is overcome, it can be cost-effective to use simpler cores, but implement more of them. For example, Intel Xeon Phi's philosophy is to have many, but smaller cores than other Xeon processors. The UltraSPARC T3 uses simpler in-order cores, but interleave instructions of many threads per core, such that the core's pipeline is filled, even when threads stall on cache misses, for example. Epiphany-IV and Exynos 4 Quad use RISC cores to reduce power usage. Interestingly, all systems either support SIMD instructions or have specific accelerators.

Table 2.2 compares the processors' performance[2]. The CoreMark [34] benchmark is used to indicate the combined performance of all cores. The benchmark tests integer and control flow performance, and minimizes the aspects of synchronization and memory bandwidth. The CoreMark score greatly differs between platforms. However, when the score is compensated for the difference in clock frequency, it suggests that the systems with more cores perform better. In this comparison, Tilera TILE-Gx8072 and Freescale T4240 perform best, Exynos 4 Quad and Intel i7-3930K worst. So, many-core systems seem to perform well, and are therefore a promising computing platform.

Following this trend, Starburst uses the simple resource-efficient MicroBlaze cores. The MicroBlaze is an in-order processor, and it is configured with a direct-mapped 16 KB instruction cache and 8 KB incoherent write-back data cache with a cache line size of 8 words, hardware multiplier, barrel shifter, and single-precision floating-point unit. In this configuration, Starburst's CoreMark score per MHz is reasonable, but still below average. However, with a score of 4521, it is close to the single-thread

[2] Unfortunately, there is no CoreMark score available for the Intel Xeon Phi.

performance of a Pentium 4 531 at 3 GHz, which is listed with a score of 5007 [34].

The peak bandwidth between the cores and shared off-chip DDR memory is also listed in table 2.2 on the previous page. The table lists the total bandwidth to all memory banks, the available memory bandwidth per core per clock cycle, and the bandwidth per CoreMark unit. Although the actual performance also depends on the rest of the memory hierarchy, these bandwidth numbers indicate a trend. Tilera TILE-Gx8072, Intel SCC, and Epiphany-IV have the lowest bandwidth per core per clock cycle, but have the highest core count. So, with increasing number of cores, the available bandwidth per core is reduced. This is the same for the bandwidth per CoreMark unit. However, the numbers are closer together, which suggests that the relatively simpler cores result in a lower CoreMark per core, and this compensates the reduction in available bandwidth.

Regarding the memory bandwidth, Starburst is equivalent to Epiphany-IV. However, compared to other architectures, the effects of the memory bottleneck will be somewhat magnified in experiments with Starburst.

2.3 Transparent processing tiles

Most architectures are shared-memory machines with multi-level caches. The hierarchy of cores and clusters is hidden behind a global address map and hardware cache coherency. This setup has several drawbacks.

The behavior of caches is unpredictable at run time [17]. Whether a cache hit or miss occurs, depends on the cache contents, which are dynamically loaded, reconciled, and flushed. A scratchpad memory *(SPM)* is a local memory next to the core, and is fully under software control. As a result, SPMs are predictable [104]. Moreover, they often give a higher performance, lower energy consumption, and lower hardware costs [85]. Therefore, SPMs are an attractive alternative to caches. However, only Intel SCC and Epiphany-IV have a 16 KB and 32 KB SPM per core, respectively. Optimal SPM allocation requires compile-time analysis of the application and efficient run-time control, which are both complex [17], or require a different programming approach.

Starburst supports both setups: the MicroBlaze caches the background memory, and also has a local memory, which can be used as an SPM, but can also be accessed by other cores. The architecture of one MicroBlaze tile is depicted in figure 2.1. The private 4 KB RAM contains the boot code, information about the system topology, and several kernel-specific data structures. The 4 KB SPM is a dual port SRAM that can be written by other MicroBlazes. Although this memory is generic, it can be used to implement core-to-core communication, such that only local memory is polled. The LMB allows single-cycle access to these memories.

The tile also contains an interrupt timer, which is used by the OS for context switching. This timer is the only interrupt source of the MicroBlaze. The statistics counters track microarchitectural events, including the number of executed instructions,

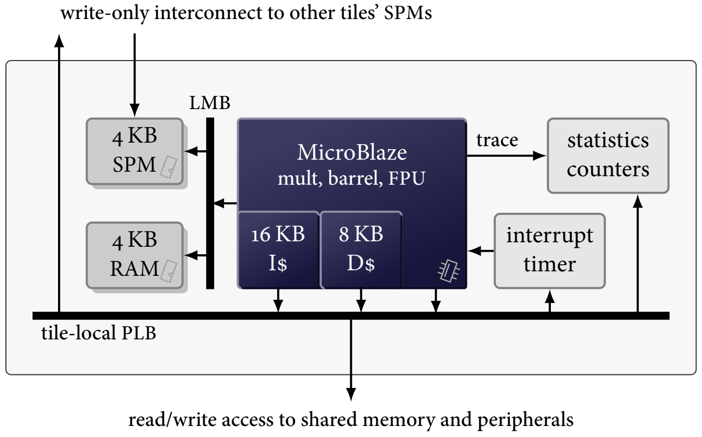


Figure 2.1 – A processing tile of Starburst. Arrows indicate master–slave relation.

cache hits and misses, and stall cycles. Because of resource constraints in the FPGA, only one MicroBlaze has these counters.

2.4 Interconnect: coherency traffic vs. data packets

Traditionally, processors connect via their own cache to a bus, which connects to the shared memory [35]. Cores communicate with this (cached) shared memory, and caches are kept coherent, e.g., by snooping the bus. So, inter-core communication is only used by cache coherency protocols; applications cannot send a specific message directly to another core, without writing it to the shared memory.

However, a single bus is not feasible when having many cores. As most architectures still support a cache coherent system, the bus is replaced by a more complex interconnect, but the purpose is still the same. Tilera TILE-Gx8072 uses a 2D-mesh NoC, where the interconnect is optimized for cache coherency and DMA transfers to peripherals. Intel i7-3930K and Intel Xeon Phi use a bidirectional ring for this purpose. Epiphany-IV does not have caches, but has a 2D mesh to access other tiles' SPMs. Intel SCC and Tilera TILE-Gx8072 expose the NoC to the application, but route the packages through the interconnect automatically. The other architectures do not specify the interconnect architecture, as it is part of the L3 cache structure.

This is different from what literature prescribes. Buses are not scalable [49], which is recognized by all architectures. NoCs are advocated as the scalable alternative [47, 114]. Most academic NoC architectures involve complex routing strategies to give timing and bandwidth guarantees on data channels in the application. The CoMPSoC multiprocessor system [51], which comprises three very large instruction word *(VLIW)* cores, a guaranteed-service NoC, and a shared memory, is an example of an academic system that follows this approach. A key property of this

Table 2.3 – Memory hierarchy properties

	type	cache coherency	memory model
Tilera TILE-Gx8072	MLC[a]	hardware	weak[b]
IBM POWER7	MLC[a]	hardware	release
UltraSPARC T3	MLC[a]	hardware	SPARC-TSO
Intel SCC	DSM	software	
Intel i7-3930K	MLC[a]	hardware	x86-TSO
OCTEON II CN6880	MLC[a]	hardware	weak[b]
Epiphany-IV	DSM		
Exynos 4 Quad	MLC[a]	hardware	weak[b]
Freescale T4240	MLC[a]	hardware, per cluster	(unknown)
Intel Xeon Phi	MLC[a]	hardware	x86-TSO
Starburst	DSM	software	slow/PMC

[a] Shared memory with a multi-level cache architecture
[b] A custom weak memory model

system is that it is composable; applications cannot affect each other's temporal behavior, because time-division multiplexing *(TDM)* arbitration is used in the NoC and in the memory controller.

However, the interconnects of the commercial architectures discussed in this chapter all have transparent arbitration schemes, and are application-agnostic. More importantly, the traffic over the interconnect is not determined by application's channels; the commercial architectures have mostly cache coherency traffic—this only relates indirectly to the communication behavior of the application. For evaluation, we follow literature, and use Æthereal [47] as interconnect initially. Chapter 4 will focus on this decision.

2.5 Weak-memory hierarchy

The memory hierarchies of the ten systems show many similarities. All systems are shared-memory architectures. Table 2.3 shows the different types of architectures, cache coherency method, and implemented memory model.

The memory model is the heart of a shared-memory multicore processor. It defines how the memory subsystem behaves in terms of state changes (writes), and how these changes are observed (reads). Sequential Consistency is a model that more or less defines that all changes to the memory are observed in the same way by all processors. In multicore systems, this is hard to implement; if two processors write into their cache simultaneously, these two changes should be communicated to all other processors in a deterministic way. To this extent, the hardware cache coherency protocol should make sure that these changes (seem to) occur atomically and instantly everywhere in the system. This is very hard to realize, and therefore architectures implement an easier, but weaker, i.e. fewer guarantees, model [17, 29]. Examples of weaker models include Release Consistency and total store order *(TSO)*[3]. As a result, a concession is made to the convenience of programming such a system. Even though all architectures claim to be programmable in C, porting software from one architecture to another is impossible, because of the different memory model.

Basically, table 2.3 shows that there are two classes of architectures: multi-level caches, and distributed shared memory. Most architectures can be classified as a multi-level cache *(MLC)* architecture: they typically have a 16 KB or 32 KB L1 cache, 256 KB L2 cache, and several megabytes of L3 cache. Cache coherency is implemented by hardware, and MLC architectures have a global address space. From a software perspective, hardware cache coherency is very convenient, as the application does not have to take control over communicating changes of the memory state to other cores—all cores 'just' see updates in the same way. However, this is not completely true. Table 2.3 lists the implemented memory models: of the eight architectures with hardware cache coherency, only the Intel architectures and the UltraSPARC T3 have clearly defined memory models, which are relatively strong. IBM POWER7 implements a model that is similar to Release Consistency. Unfortunately, we cannot find specific details about the memory model of Freescale T4240. The others do not clearly define the model, besides stating that it is 'weak'.

Table 2.3 confirms the expected trend towards weaker memory models, but architectures still implement hardware cache coherency. However, this is also expected to change. As the density of DRAM increases, e.g., by 3D die stacking, locality becomes increasingly important [22]. Therefore, changes to the memory are likely to be kept local, resulting in incoherent (clusters of) distributed memories. Moreover, hardware cache coherency has a significant overhead [29]. Although software cache coherency is more complex to use, it outperforms hardware in terms of performance and energy usage [5]. Additionally, domain-specific architectures typically omit coherency at all [17], or leave the shared memory uncached [91].

Summarized, it is expected that future multicore hardware will only implement a weak memory model, without cache coherency or only software cache coherency. Two architectures of table 2.3 adopted this approach: Intel SCC and Epiphany-IV, which are distributed shared memory *(DSM)* architectures. These architectures have a NUMA partitioned global address space, where specific memory regions are local to a core. Coherency is only manually realized, as the application has control over the communication of data between local memories.

Even though a weaker memory model is used, shared memory stays the dominant memory architecture. In literature, it is argued that the hardware platform should preferably support shared memory to reduce the programming effort [36, 67]. Namely, other architectures, such as a streaming setup and message passing, can be emulated on such a system by means of a software middleware layer [35, 36, 67, 111].

[3]Section 3.2 and chapter 5 discuss memory models in more depth.

Starburst follows these trends: as it does not have hardware cache coherency, it is configurable either to have the shared memory uncached, or to use software cache coherency. Chapter 5 discusses handling such a memory architecture in detail.

2.6 Programming model consensus

The ten commercial architectures are all marketed as powerful hardware architectures within their domain—details on how to program them are very sparse. At least, all architectures support C using the `gcc` and `binutils` tool chain, complemented with debugging features around `gdb` and a graphical profiler. Concurrency is in principle realized using threads, although other models can be implemented on top, like OpenMP.

The Intel SCC runs many stand-alone Linux kernels, and requires the programmer to handle cache coherency and distribution of data. The Intel Xeon Phi has several programming models, including using all cores as coprocessors for function offloading. All other architectures claim to run an SMP Linux version. UltraSPARC T3, Intel i7-3930K, and IBM POWER7 clearly define the memory hierarchy and properly support Linux. The others fail to mention any shortcomings. It is unknown whether thread migration and load balancing is supported, as it is an expensive and possibly complex task. Moreover, dynamically balancing threads neutralizes any benefit of locality, which is a key aspect of these architectures.

If we assume that Pthread [94] is used as threading model, synchronization of data is under control of the programmer. Usually—but not necessarily—a Pthread mutex, condition, or barrier is used to protect shared data. However, as Pthread does not prescribe binding a synchronization variable to the shared data it is related to, the OS does not have any knowledge about which data is to be synchronized and communicated to other cores. Since Tilera TILE-Gx8072, OCTEON II CN6880, and Exynos 4 Quad only support a weak memory model (see table 2.3 on page 16), it is unknown how and when data is communicated, without additional effort of the programmer. And which effort is required, is not (publicly) documented.

Epiphany-IV is programmed using ANSI-C. In contrast to the other systems, it does not run Linux. The cores do not have caches, and data from main memory is fetched to a local memory using DMA transfers. Therefore, it cannot execute code from main memory directly. As ANSI-C does not support threads by the language itself, it also requires a library like Pthread to start threads on other cores. However, the same drawbacks apply, regarding distribution and communication of data as discussed for the three architectures above.

Although the architectures discussed above use processing tiles to exploit locality, no architecture advocates using a programming model that matches this setup. All systems use threading and shared memory, which only has limited support for locality and core-to-core communication.

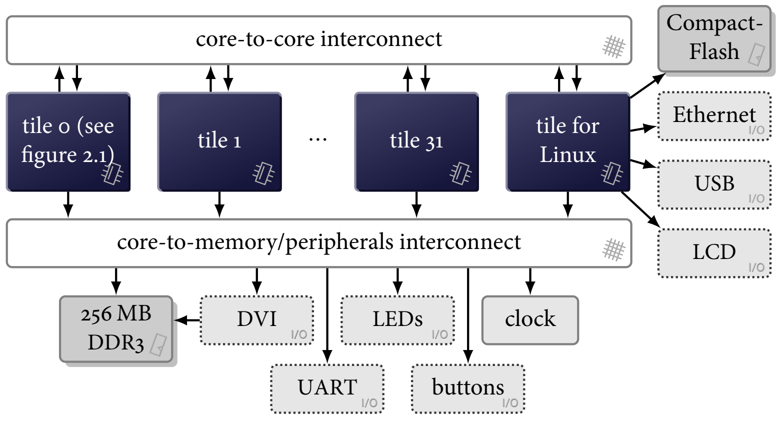


Figure 2.2 – Starburst system overview

2.7 Starburst

Throughout this thesis, we use Starburst as evaluation platform, which is a many-core NUMA DSM architecture, with a weak memory model, without hardware cache coherency. As discussed above, the aforementioned trends are reflected in this system. The overview of the whole system is depicted in figure 2.2.

2.7.1 System overview

The system contains up to 32 MicroBlaze tiles (see figure 2.1 on page 15), and one additional tile that is reserved for Linux and several peripherals. This number is configurable, but limited by the available resources of the FPGA. All cores can write into each other's local memory via the upper interconnect, therefore the topology of this interconnect is all-to-all. The DDR memory can be accessed via the lower interconnect, which arbitrates and multiplexes memory requests from all cores to one memory port.

The system has several peripherals: a DVI port with a resolution of 640×480 32-bit pixels, UART, several LEDs, and buttons. A counter keeps track of the current time. As the bandwidth requirement between MicroBlazes and these peripherals is very low, they share the arbitration interconnect with the memory. For more complex peripherals, the Linux tile is included. This tile has a similar layout as all other tiles, but contains PLB slaves to access Ethernet, USB, and a 2 GB Compact-Flash memory card. The currently used Linux kernel version is 3.4.2, which has its (bootable) file system on the CompactFlash card. It runs a Telnet service, and it allows accessing for example a memory stick, keyboard, and headphones via USB. Via Ethernet and Linux, programs can be uploaded to the main memory, and Linux can bootstrap all other MicroBlazes. The Linux core is turned off during all performance experiments, to prevent interference on the memory interface.

We implemented a tool flow around Xilinx Platform Studio, which takes a high-level architecture description and generates a Starburst instance. The Xilinx Virtex-6 FPGA ML605 Evaluation Kit [117] is used, which contains a XC6VLX240T FPGA.

2.7.2 OS: Helix

Every MicroBlaze runs its own instance of our OS, called Helix. Helix has a microkernel, which takes care of scheduling, and process creation and cleanup. All other functionality is implemented by daemons. From kernel perspective, daemons are ordinary processes, but they implement a crucial part of the functionality of the OS. Processes are always bound to a specific processor and cannot migrate. Every kernel runs a synchronization daemon, and a daemon for heap management (for `malloc()` and friends). Upon request, other daemons are available to handle timers (via `timer_create()`), and to gather process utilization statistics.

The synchronization daemon dispatches messages between processes of different tiles. This is done via the SPMs within every tile. In this memory, FIFOs are allocated, such that there is a channel between every pair of cores. The synchronization daemon multiplexes messages of all local processes over these channels in a round-robin fashion. The types of messages include process management on a remote core, inter-process signals, and lock messages (to be discussed in section 4.5). Messages are up to six words in size. Processes can also directly allocate memory in the SPM and implement FIFOs on top for inter-core communication, without any control of the synchronization daemon.

All processes and system calls are preemptive. To maintain a predictable performance, Helix's system calls never lock a mutex to prevent problems like priority inversion [95], and Helix runs a budget scheduler. In such a scheduler, every process gets a budget, i.e. time slice, assigned. So, the balance between slices determines the amount of processor time every process gets. Helix schedules in a fixed order, and processes run until their time slice has ended, or they voluntarily yield the processor.

A fixed-order schedule of processes with time slices is equivalent to TDM. In such a schedule, a *replenishment interval* denotes the period of time in which all processes are executed for the length of their time slice. After this period, every budget is replenished with their initial amount, and the scheduling starts over. If an incoming message arrives for a process that just finished its time slice, the message can only be handled in the next replenishment interval. Since a daemon process is used to dispatch messages between processes, the length of the replenishment interval has an impact on the latency of messages, and possibly on the performance of the system. Therefore, we modified the TDM scheduler to be able to let the synchronization daemon interrupt any other process when a message arrives, as long as the synchronization daemon has a time budget left. This scheduler is similar to the priority-based scheduler [100], where the synchronization daemon is the only high-priority task, which can interrupt all other (low-priority) tasks. To this extent, the scheduler works with slots of about 0.6 ms, controlled by the interrupt timer. Dur-

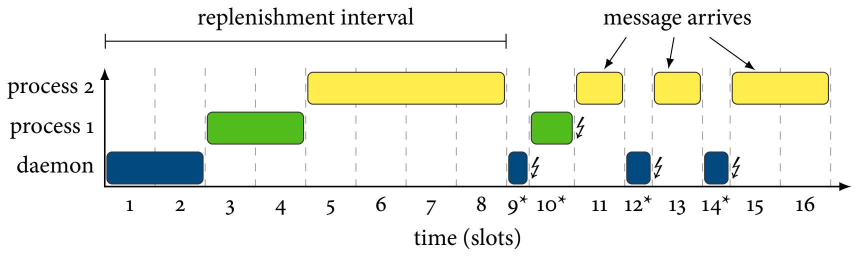


Figure 2.3 – Scheduling example of Helix

ing an interrupt, it checks whether messages are available. If so, a context switch is done to the synchronization daemon. Moreover, the interrupt handler decrements the remaining budget of the currently running process. When the budget becomes zero, the next process is scheduled. The overhead of checking for messages, updating the budget, and immediately continuing the currently running process is 12 clock cycles at most.

Figure 2.3 exemplifies a schedule as implemented in Helix. During the first replenishment interval, every process runs until every allocated slot, i.e. the complete time slice, is used. In the second interval, the synchronization daemon and process 1 yield the process prematurely, which is indicted by a ↯. When a process yields, it triggers an interrupt, such that the current slot is ended immediately—shortened slots are indicated in the figure with a *. Although process 1 has some budget left, it will not be scheduled until the next replenishment interval. However, the synchronization daemon is handled differently, and it is allowed to handle incoming messages. In the example, three messages from another core arrive in one of the FIFOs in the local SPM during the execution of process 2. The first two will be serviced by the daemon after the next interrupt. The daemon will yield as soon as there are no more messages to process. The third message has to wait until the next replenishment interval, since the daemon has used its entire time budget for this interval. The figure shows that process 2 is not interrupted after the arrival of the message in slot 15.

Table 2.4 on the next page gives an overview of the memory regions. All code resides in DDR memory, and all MicroBlazes execute it from there, except for the scheduling code, which is copied to the local RAM upon kernel initialization to reduce scheduling overhead. Every Helix instance manages its own local heap of only 3 MB, which is also used to allocate all processes' stacks. Because this heap is private for a MicroBlaze, no cache coherency is required. The local SPM also contains a heap structure, which is managed by the local MicroBlaze, but writable by others. All shared data resides in DDR memory, which is either statically allocated or also organized as a heap data structure. Shared data is either uncached, or cached with software cache coherency, which is discussed in detail in chapter 5.

Table 2.4 – Helix memory layout

memory	section	size
4 KB RAM	interrupt vectors, boot code, system topology	256 B
	scheduler code/data	2 KB
4 KB SPM	synchronization daemon data structures	≈200 B
	heap, accessible by other cores	the rest
128 MB DDR, cached	kernel/application (`.text`)	usually <1 MB
	read-only data (`.rodata`)	application specific
	per-MicroBlaze process table and heap	3 MB heap per core
128 MB DDR, uncached or software coherent	static variables (`.bss` and `.data`)	application specific
	shared heap	96 MB
	reserved for Linux	16 MB

2.7.3 Application environment

From application perspective, Helix offers a POSIX-like environment. Helix supports the newlib C library, and implements the Pthread standard. Additionally, POSIX signals and timers are available. C using `gcc` is fully supported, only proper thread-safe C++ exception support for `g++` is lacking, as it requires MicroBlaze-specific threading support in the compiler and support libraries, and none of our application requires it.

To utilize the SPMs effectively, an API maps FIFOs on them. Such a FIFO can be of any depth and element type. It uses the C-HEAP communication protocol [86], which uses a duplicate administration at both the sender and receiver side. During communication, only the local (conservative) administration is read or polled, where posted writes are used to communicate data and update the administrative fields at the other party.

Helix itself does not have a file system. However, all standard file operations are forwarded to a userspace daemon running in Linux. Therefore, all MicroBlazes can access the full file system, including device nodes, named pipes, etc. Among other tasks, keystrokes of the USB-keyboard can be read, and files on an NFS mount can be accessed.

2.8 Benchmark applications

All architectures discussed in this chapter are general-purpose, and focus mostly on high performance. However, as processors become faster, they can be used for tasks where ASICs were used before. For example, software-defined radio is an application of programmable mobile architectures that replace analog circuits and ASICs [2]. To this extent, most architectures include support for vector or SIMD instructions, which are suitable for digital signal processing *(DSP)* applications.

The trends of moving towards incoherent caches or SPMs, as discussed above, puts pressure on programming such a system. Communication within a chip is increasingly under software control [22]. Moreover, an SPM has a limited size, which makes the notion of locality in the software more important. This requires disciplined programming models to cope with the additional programming complexity [29].

On one hand, there is a push towards concurrency, locality, and disciplined programming from the hardware manufacturers. On the other hand, more DSP applications emerge, like software-defined radio and multimedia processing, which are parallel in nature. The combination leads to the concept of *streaming applications* [106]. A streaming application is usually modeled as a Kahn process network *(KPN)* derivative, where actors communicate via channels to each other. This concept fits nicely to the trends regarding concurrency (by actors) and locality (by explicit data movement in FIFO buffers), and hides the complexities of managing the cache or SPM.

There is a catch. A KPN assumes infinite buffer sizes, which makes them impractical to implement. In general, the maximum buffer sizes of an application that is based on a KPN, cannot be calculated. Other models, e.g., SDF and CSDF, are more restrictive, and therefore better predictable. However, in contrast to a KPN, these models are not expressive enough in general [90]. Hence, any practical application might be limited to specific cases.

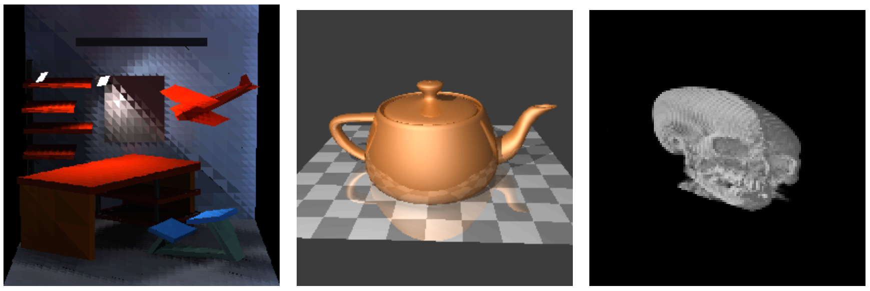
According to Culler et al. [35], message passing handles memory replacement less dynamic than shared memory, which can result in more memory overhead. Shared memory is more generic, as it can emulate other models, including message passing. So, regardless of how an application is modeled, a shared-memory architecture for hardware is likely to sustain.

To the best of our knowledge, no streaming application benchmark set exists. Because shared memory is still a relevant model, we use several shared-memory Pthread-based parallel benchmark applications. These applications are discussed next for future reference.

2.8.1 SPLASH-2

From the SPLASH-2 [115] benchmark set, we use three applications: `radiosity`, `raytrace`, and `volrend`. Every application makes use of worker processes, which all do a part of the work. Every MicroBlaze runs exactly one of such a process. Screen dumps of these applications are shown in figure 2.4 on the following page. The applications are used in chapters 4 and 5.

`Radiosity` splits a 3D model of a room into small triangles. Then, the interaction of the luminance of all pairs of triangles is calculated, starting from the lamps, which illuminate the otherwise dark room. Iteratively, all pairs are processed, until the distribution of light stabilizes. The program exhibits chaotic memory accesses, which result in much synchronization and cache coherency traffic. Drawing the output,



(a) radiosity (b) raytrace (c) volrend

Figure 2.4 – SPLASH-2 applications

as shown in figure 2.4(a), is not part of the benchmark itself, and is an optional step afterwards. However, it shows an often-occurring bug: only the bottom-left part of the square in the middle is lit—the output is somewhat non-deterministic.

Next, `raytrace` renders a teapot, including reflections. The rendering is done in a single pass. It splits the work in packets of 8×8 pixels, which are distributed among all worker processes. The default data type for all computations is a `double`. As the MicroBlaze has only hardware support for `floats`, this is a performance penalty. Interestingly, when all `doubles` are replaced by `floats`, the output is bit-by-bit identical, but the performance increases almost by a factor of four. For all experiments, the original code with `doubles` is used.

Finally, `volrend` draws a skull, based on a 3D model out of voxels. Similar to `raytrace`, the rendering is split in packets of a fixed amount of output pixels, which are drawn concurrently. The program shows an animation of a rotating skull.

2.8.2 PARSEC

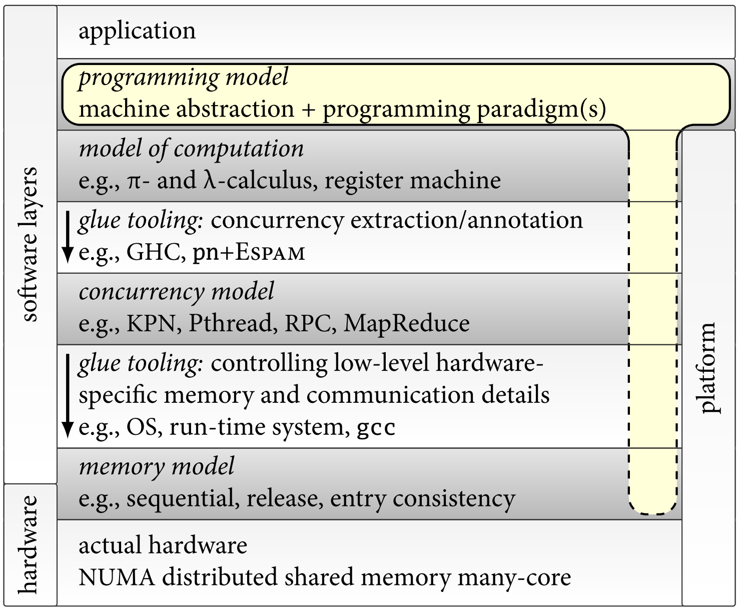
From the PARSEC benchmark suite [14], only `fluidanimate` is used. This program does a particle simulation. Similar to `radiosity`, the interactions between pairs of particles are calculated, which result in a chaotic memory access pattern. This application is used in chapter 4.

Of all other applications of the benchmark suite, only `blackscholes` is runnable. However, it is of limited use, as it does not use synchronization between worker processes. The other applications are not compilable, because of dependencies on libraries that are not portable to non-standard platforms such as a MicroBlaze and Starburst, or are not runnable, because of memory constraints.

2.8.3 NoFib

In chapter 6, concurrency is evaluated using functional languages. We implemented a functional language on top of Pthread for x86 systems and Helix. The details of the implementation are discussed in that specific chapter. The applications we use are taken from the Haskell NoFib [88] benchmark suite. For completeness, we briefly discuss the applications below.

`Coins` counts the number of ways one can pay a specific amount of money, with a given set of coins. `Queens` determines in how many ways n queens on an $n \times n$ chessboard can be placed, such that they cannot capture each other. `Parfib` calculates the Fibonacci number of a given point in the sequence in an inefficient, but parallel way. `Partak` is a parallel version of the recursive Tak function τ, which is designed to have an unpredictable amount of remaining work in each step of the computation. `Prsa` encrypts a string of tokens using the RSA algorithm.


application
programming model
machine abstraction + programming paradigm(s)
model of computation
e.g., π- and λ-calculus, register machine
glue tooling: concurrency extraction/annotation
e.g., GHC, pn+Espam
concurrency model
e.g., KPN, Pthread, RPC, MapReduce
glue tooling: controlling low-level hardware-specific memory and communication details
e.g., OS, run-time system, gcc
memory model
e.g., sequential, release, entry consistency
actual hardware
NUMA distributed shared memory many-core
software layers
hardware
platform

Chapter 3 overview

3 Platforms and Programming Abstraction

Abstract – *A many-core platform inherently needs concurrency. This property has a major influence on how it should be programmed. Because of parallelism in the execution, several models become prominent: the behavior of the (distributed) memory determines how communication is realized, as defined by the memory model; the concurrency model defines composition and interaction of concurrent computation; and the model of computation defines the fundamentals of the algorithm.*

Taking full control over all details of these models is almost impossible for the programmer to do by hand. A programming model presents all layers of the underlying platform in a convenient way. Different programming languages make different trade-offs for this model regarding the level of abstraction, ease of programming, and control over the hardware.

Chapter 1 discussed the relevance of programming for concurrency, and mentioned the necessity of abstraction. Concrete architectures were discussed in chapter 2. As stated before, all these architectures can be programmed using C. Apparently, the architectures can be abstracted in such a way that a single programming language is suitable to be used on all of them. This chapter will discuss the abstractions of 'programming' and their relation.

Consider the example program in C pseudo-code of two parallel executing processes in listing 3.1 on the following page. The intention is to communicate the value 42 from process 2 to 1. The variable `flag` is used to synchronize the processes and to signal that `X` has been written. Although this code looks fine at first glance, many assumptions are made, based on the idea the programmer has about the machine. The way of thinking about the machine depends on many abstract models, which are briefly discussed next and defined in subsequent sections.

Process 1:

```
flag = 0;
while(flag!=1) {}
r1 = X;
```

Process 2:

```
X = 42;
flag = 1;
```

Listing 3.1 – Polling a flag

There is a notion of the abstract machine. In this case, it is a parallel machine, which allows executing two independent processes. Every process is a sequence of steps, and every step modifies the state of the machine. Which kind of steps are allowed and how they contribute to the result is defined by the *model of computation*.

In this example, multiprocessing, i.e. threading, is used to deal with concurrency. Processes can access the same variables, and use a central shared memory to communicate data. This is in contrast to a dataflow machine, which communicates via FIFOs. How concurrent composition and interaction is defined, is part of the *concurrency model*.

The example uses variables, which are shared between processes. In C, one often assumes that writes of variables are executed atomically and instantly. However, as processes can be executed on physically separated processors, updates might take some time to propagate through the system. The behavior of the memory subsystem is captured by the *memory model*.

As we will see in a moment, these models are often not completely independent. Moreover, hardware architectures have a greater influence on how software uses memory models than on what programming paradigm should be use. So, there is an order in which the actual hardware determines the freedom of choice by the application (programmer). We will discuss the models in this increasing order of freedom. The figure on page 26 visualize the organization of the models as a stack, having the hardware at the bottom. In the course of this chapter, this organization is explained, combined with definitions of the models, and a discussion of the layers in between.

3.1 Many-core hardware is the driving force

The stack of models has two ends: the hardware and the application. There is a trade-off what can be done in hardware, and what can be done in software (as part of the application). One extreme is to do everything in hardware; this is an ASIC. The other extreme is to do everything[1] in software. One can imagine that this involves that software controls low-level events, like bus arbitration, NoC routing protocols, and keeping track of the required refresh cycles of DRAM.

[1] In contrast to ASICs, which are hardware without software, software cannot run without hardware. Even in virtual environments, there is some piece of hardware executing the software in the end. What the fundamentally minimal required hardware is, remains an open discussion.

In common systems, programming is made relatively easy, at the expense of hardware complexity. To this extent, hardware implements floating-point arithmetic, cache coherency, arbitration on shared memory, atomicity of state changes, and support for synchronization and context switching. All of this can be done in software, but is easier and faster in hardware. It should be clear that hardware is bound to physical properties and fundamental limits, where software is an abstraction that can be changed freely. Fundamental boundaries of the hardware are approaching, as the energy consumption is limited by heat dissipation, data movement at high clock frequencies by the speed of light, and transistor size by the size of atoms. Trying to match the hardware architecture to what is convenient to have in software, is becoming harder than matching the software model to the hardware properties.

To be concrete, the most likely usage of the additional transistors is to increase the number of cores—as discussed in chapter 1. So, the hardware architecture determines the need of concurrent programming. Although this complicates programming, this has been accepted as a change in the programming model. The trends in many-core architectures, as of chapter 2, also drive changes in programming; a weaker memory model or the lack of cache coherency strongly influences programming such architectures. Therefore, the hardware is the most influential layer in how a multiprocessor system can be programmed. What is possible to realize in hardware influences how abstraction layers on top of it can be defined.

The trends in hardware force us towards concurrent many-core programming, even though software becomes inherently complex because of it. This yields two important questions: how do calculations by all individual cores contribute to the overall computational problem, and how do they communicate intermediate results? Choices about the latter are solely determined by the hardware at hand, and the former is mostly a software choice about how concurrency is organized. As the memory model defines how communication of data (via memory) behaves, it is more closely related to the hardware, and therefore lower in the stack of models. We will discuss the memory model next. How concurrency is modeled is discussed in section 3.3.

3.2 Memory model—the hardware's contract

Cores in a many-core DSM architecture communicate via shared memory. Since the memory is usually an off-chip DDR memory bank, transactions take time to complete. The combination of parallelism and transaction latency makes it hard to define a proper state of the system, as there does not have to be an ordering relation between multiple events, i.e. transactions. Which ordering is defined, and therefore, how the memory behaves, is part of the memory model [4, 35]:

memory (consistency) model

> *An abstraction that defines the constraints on the order in which memory operations must appear to be performed, i.e. become visible to the processors, with respect to one another. It defines the semantics of shared variables.*

Consider the example of listing 3.1 on page 28. In this example, there are two shared variables, namely `flag` and `X`. Two different types of memory operations are applied on them: reads and writes. The definition of the memory model states that constraints are defined. An example of such a constraint is: "Writes of one process keep their order." So, the writes at lines 4 and 5 are observed to be in that specific order, by all processes. Based on the guarantees that can be inferred from this constraint, synchronization can be implemented by polling the `flag`, as shown in the example. Although this might sound trivial, this has an impact on the hardware that implements these constraints. For example, if a 2D-mesh NoC has multiple routes from a core to memory, it still has to guarantee that the first write arrives at the memory before the second write.

Different memory models define a different set of constraints. Extreme examples of memory models are Atomic Consistency, which specifies that all operations occur instantly in a globally observable total order, and Slow Consistency, which does not define any constraint, except that only the data dependencies locally to the executing core are preserved. Notably, the Sequential Consistency *(SC)* [70] model is often used as the reference model. SC states that all operations are observed to all processes as if they were executed by a single process. Informally, this is the 'natural' view on memory: all reads and writes behave as one would expect intuitively how a single memory would behave. Hence, it is easy to understand and therefore preferable to implement [55]. A model that prescribes many constraints is often characterized as a *strong* or strict model, opposed to *weak* or relaxed models, which enforce less ordering and therefore allow more non-determinism [83, 101]. Chapter 5 discusses memory models in depth.

The memory model can be seen as a contract between hardware and software. Software relies on the semantics of the memory as defined by the model, and the hardware implements that model. Hence, *all* hardware components are subject to the memory model, not only the memory and the interconnect. Even the core itself is relevant, when it exhibits out-of-order execution, for example. In that case, the two writes are initiated in the order as prescribed from process 2's perspective. However, out-of-order execution might result in writing `flag` to background memory first, so process 1 can observe these write operations in reversed order.

'All hardware components' also include caches, although cache coherency is generally considered to be a separate problem. Caches influence the behavior of memory operations, just like the interconnect. Therefore, it should not be the case that the cache coherency protocol *determines* the multiprocessor consistency model, as stated by Stenström [102]. Instead, the coherency protocol should *follow* or implement the requirements that are imposed by the intended memory model of the architecture[2].

Most architectures of chapter 2 have caches. A strong memory model, which might define a constraint such as: "All writes to the same (shared) variable must be in total

[2] This relation between consistency and coherency does not necessarily make coherency protocols easier to implement.

order", is hard to realize by a cache coherency protocol. Consider the case where two cores—separated by a multi-hop NoC—write to the same variable at exactly the same time. Then, a total order cannot easily be enforced, as updates propagate slowly through the system. As a result, architectures use a weaker model [4], such that the hardware complexity is reduced, and the system is more scalable regarding the number of cores. This corresponds to the trend as observed in section 2.5.

At application level, it is convenient to have a strong memory model. When the memory model that is implemented by the hardware, is weaker than required, software can *complement* the hardware's memory model. This is the case for software cache coherency; when caches are not kept coherent in hardware, software can take over the coherency protocol to realize it at a higher level. To this extent, the hardware should, for example, have support for cache flush instructions—to realize software cache coherency properly, co-design is required to match memory model requirements of the application and the features of the hardware. This is also worked out in detail in chapter 5.

3.3 A concurrency model to orchestrate interaction

To utilize the computational capabilities of a parallel machine, the software has to use some form of concurrency. How concurrency can be achieved, is defined by the *concurrency model*. Although literature does not properly define such a model, we use the following explanation:

concurrency model
: *An abstraction that specifies how computation is decomposed into concurrent components, and defines rules for interaction between them.*

Threading is probably the best-known concurrency model. A concrete threading model is Pthread [94], which clearly defines decomposition in terms of threads, and interaction in terms of shared memory that is synchronized by using a lock, condition, or barrier. Pthread does not enforce how the computation itself is done by a thread, and leaves freedom in the underlying memory model. The synchronization primitives allow a translation to practically any memory model, by a compiler or a library. Although Pthread can be considered as a concurrency model, it does, however, put some restrictions on the memory model. For example, it has to have a shared address space, and cannot be used upon a message-passing architecture that does not have global memory addresses. This exemplifies that this specific concurrency model has influence on the choice of a memory model.

A Kahn process network *(KPN)* [65] can also be considered as a concurrency model. In its original form, it is based on sequential processes written in Algol, extended with functions to send and receive data via unbounded FIFO channels. There is no notion of memory addresses. As a result, any underlying (shared) memory model can be used, as long as FIFO channels can be modeled upon it. Kahn describes a KPN as a parallel programming language, but mostly focuses on how these processes interact in terms of abstract functions and their input–output relation, regardless of scheduling. The KPN formalism does not include the semantics of what happens inside a 'computing station', and only defines program structure, communication channels, and synchronization. This also holds for KPN derivatives, like dataflow, SDF, and CSDF. As the actual computation is not part of the formalism, a KPN and its derivatives cannot be considered as a model of computation. This conclusion does not match dataflow literature [72].

Another example of a concurrency model is MapReduce [38]. It defines that a (very large) data set is split in chunks, a function is mapped upon all of them, and the intermediate results are combined into a smaller set of output data. Because the computation during every individual map and reduction function is limited to a small chunk of data, locality is easily exploited, and concurrency is realized. This makes the model appropriate for large (distributed) data centers. Dean and Ghemawat [38] claim that MapReduce is a programming model, but as the model, like KPN, focuses on (de)composition of data and tasks and does not define computation itself, it fits our definition of a concurrency model better. In contrast to Pthread, MapReduce does not assume shared memory, which makes it independent of the underlying memory architecture.

Many other concurrency models exist. One can think of a remote procedure call *(RPC)* as a way to exploit concurrency in a client–server setup of multiple computers. Furthermore, function offloading can be used to utilize accelerators within a system-on-chip. Moreover, instruction-level parallelism *(ILP)* transparently uses parallel components within a core or ALU. Also, a systolic array and vector processing in a SIMD processor or GPU are classified as concurrency models.

In the example in the beginning of this chapter, threading is assumed as concurrency model. A part of the model is that progress of the threads is not guaranteed; process 2 might finish before process 1 even starts. Because of this, there exists a race condition: if the write at line 5 is executed before line 1, the synchronization fails. Such a non-determinism in the order of execution gives freedom in the implementation of the hardware, but makes debugging hard.

In contrast to the memory model, how concurrency is organized is more or less a software view on the parallel hardware. For example, multithreading and processes are software concepts and are handled by the operating system, with help of the underlying hardware. When the underlying hardware is a shared-memory architecture, all communication patterns can be realized—although the efficiency can differ. This makes the concurrency model independent of the implemented memory model. Therefore, it is higher in the stack of models, as depicted by the overview figure on page 26.

The transformation of the concurrency model into something that is runnable on hardware, can be done by either run-time or compile-time software. For example, the operating system dynamically handles context switching between processes. Moreover, worker threads schedule tasks of Apple's Grand Central Dispatch [9] or sparks of Haskell. An assembler can make decisions about insertion of fence instructions when data is communicated, depending on the actual memory model. For a KPN, the channel implementation could control cache coherency, when appropriate. Everything that implements the concurrency model on top of the architecture is *glue tooling*.

3.4 Computation and algorithms

The concurrency model defines the interaction of components that compute, but it is undefined how computation itself is realized. For this purpose, we need a *model of computation*[3].

This term, however, is often used without a proper definition. In the Ptolemy II [23] project, models of computation are evaluated. The project defines such a model as "the rules that govern the interaction, communication, and control flow of a set of components". However, this definition lacks how 'computation' is realized, and matches our definition of a concurrency model better. Skillicorn and Talia [99] give the following definition: "A model of (parallel) computation is an abstract machine providing certain operations to the programming level above and requiring implementations for each of these operations on all of the architectures below." Programming is included in this definition, but a programming model is fundamentally different view on the abstract machine, as we will see in section 3.5. Finally, Wikipedia states: "A model of computation is the definition of the set of allowable operations used in computation and their respective costs." We use the following definition:

model of computation

> *An abstraction that defines the elementary operations, i.e. transformations, for computation. Such a model allows expressing algorithms, which are sets of transformations, and data, which is used to apply algorithms on.*

A Turing machine is a well-known model of computation. In such a machine, a computational problem is formulated by the state table (the algorithm), and a sequence of symbols on the tape (the input data). Figure 3.1 on the following page depicts a very simple algorithm that adds two numbers, which are coded as the length of a sequence of 1's. The input is a tape with two numbers, 2 and 3 in this example, separated by a 0. When the machine halts, the result is on the tape.

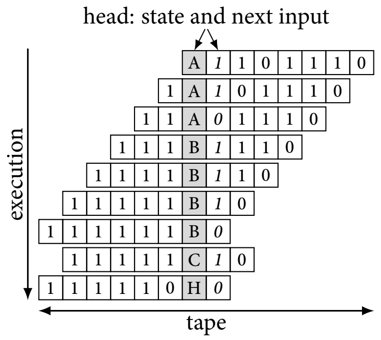
Similar to a Turing machine, the (abstract) register machine can also be positioned as a model of computation. Notably, the random-access stored-program machine *(RASP)* model is a Von Neumann architecture, although with infinite number of registers [33]. Modern processors are based on this model, and extend it for performance purposes with instructions beyond load/store, add/subtract of integers,

[3] A model of computation should not be confused with a *computational model*. The latter is a (mathematical) model of a complex system, which is used in a simulation environment. Just an example for clarification: the computational model of the weather is used for rain and temperature predictions.

	tape input	
state	0	1
A	1→B	1→A
B	0←C	1→B
C	0→HALT	0→HALT

1→B means: write 1, move head to right, go to state B

(a) Add algorithm



(b) Execution steps to add 2 and 3

Figure 3.1 – Simple add algorithm for a Turing machine

read/print, and a conditional branch. These models of computation do not explicitly enforce a specific concurrency model, but it is far from trivial to map an algorithm for one of these machines to an arbitrary concurrency model. The example of listing 3.1 on page 28 assumes a parallel RASP model, which has two parallel executing sequential register machines.

A different model of computation is λ-calculus [30]. In this model, computation is defined by means of functions, and a rule to reduce a function application to its result. Chapter 6 discusses this model in detail. In contrast to a Turing machine, there is no specific order in which function applications must be evaluated. As a consequence, λ-calculus naturally allows the implementation to apply reductions of an algorithm in parallel. Where λ-calculus is built around functions, π-calculus [80] (or process calculi in general) defines operations on processes and channels. Although the model might be implemented in different ways, a naive implementation could map an algorithm in π-calculus rather straightforward to the dataflow concurrency model.

All models of computation discussed above are Turing complete, and can therefore emulate each other. Hence, the choice for the model of computation that is used in the application, and for the one that is implemented by the hardware, does not necessarily have to match—although there might be some overhead in the translation. In the overview figure on page 26, the model of computation is positioned above the concurrency model, because the choice of it (mostly) depends on the programmer. Similar to the glue tooling between the concurrency and memory model, the overview figure also describes glue between the model of computation and the concurrency model. This tooling is responsible for translating (parts of) the algorithm to the proper way of concurrency. For example, the Glasgow Haskell compiler *(GHC)* [46] compiles expressions in λ-calculus, which is basis for a functional language, to atomic units, which in turn are distributed among worker threads for computation. Additionally, pn [112] is a tool that extracts a parallel process network, which is similar to dataflow, from a sequential input program, which is known as a static affine nested loop program *(SANLP)*.

3.5 Programming model: a peephole view

Although the memory model, the concurrency model, and the model of computation are stacked in the overview figure on page 26, they are not layers with an increasing abstraction. Instead, the layers have orthogonal properties, and are ordered by the freedom of choice by the programmer. Some details of these models can be handled automatically, some must be controlled by the programmer. For example, if a KPN is used as concurrency model on top of a shared-memory system, all memory model aspects can be handled automatically by the implementation of channels. So, abstractions in a higher layer can hide details of layers below, such as a KPN channel library can easily transparently insert cache flushes, when required. The stacking order of the layers therefore indicates in what order layers can be hidden from the programmer.

The choices for specific abstractions influence how a programmer sees the system and what it takes to write an application. A *programming model* captures all aspects of all abstractions of the hardware, and presents only relevant parts of the system to the programmer, leaving all details out that the tooling can handle by itself. We use the following definition:

programming model

> *An abstraction of the system that consists of an abstract machine, a programming paradigm, and a subset of features of the underlying models that has to be dealt with by the programmer.*

A (good) programming model is tailored towards convenient usage by the programmer. The abstract machine is closely related to the machine of the model of computation. It is the general concept of the system that is programmed. For example, an abstract machine as a 'multithreaded core' has the concept of a register machine, running threads in parallel—even though the OS implements context switching, and the hardware might not have any knowledge of threads. The programming paradigm is a way of organizing the application. The model of computation can be reflected in the paradigm, such as the functional or imperative paradigm. It can also include compositional aspects, like in the object-oriented paradigm.

3.5.1 Existing programming language's models

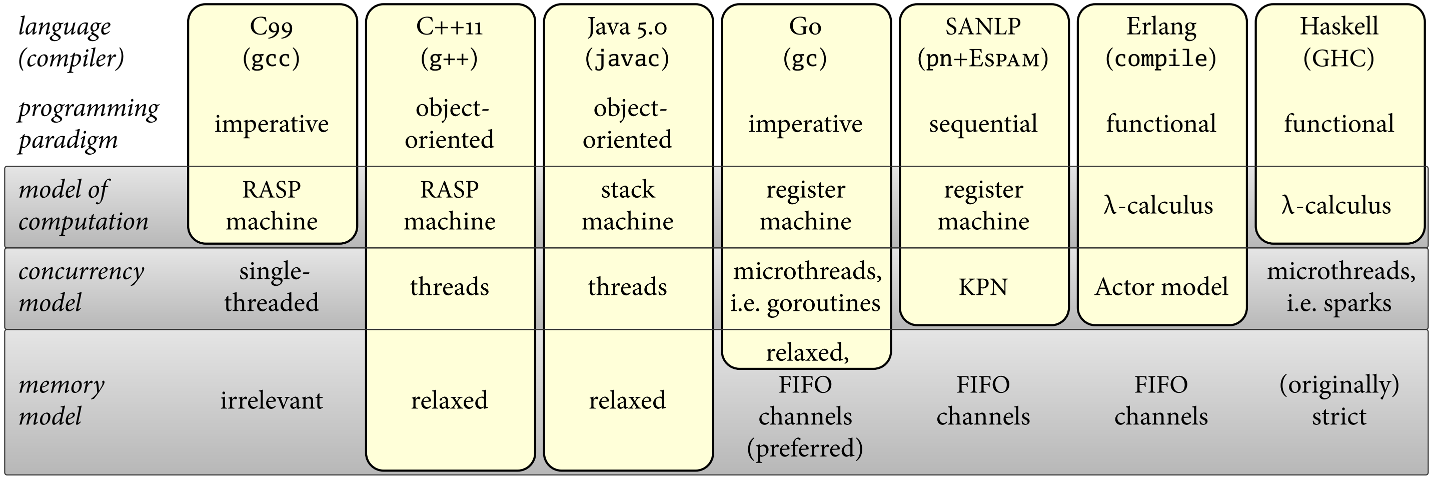
The definition of the programming model includes 'features of the underlying models', which we will explain based on existing programming languages. A programming language can be seen as an instance of the programming model, complemented with syntax and a type system, for example. As a programming model usually does not have a name, and the exact syntax, parsing, and type system is not relevant for this thesis, we use the language's name to address its programming model. Figure 3.2 presents programming languages and properties of the models they are based on.

The first language in the figure is C99 [25]. The language exhibits an imperative or structured paradigm, and the abstract machine belongs to the class of register machines. Because the language is single-threaded, memory consistency is not relevant. Therefore, the programming model only defines a fairly simple sequential machine. The fact that the C99 programming model does not include the memory model, means that the compiler can handle all machine-specific memory issues, such as alignment, and memory-related optimizations, like redundant load elimination. Although there are threading libraries for C99, such as Pthread and OpenMP, these are not part of the language. As a consequence, any memory model that comes with these libraries, defines amendments to the language, and a concurrency-agnostic C99 compiler might break the program. Boehm [20] argues that adding threads to C by libraries is flawed by design, because of this reason.

C++11 [24] is object-oriented and uses a similar model of computation as C99. Additionally, it defines a threaded concurrency model that communicates via shared memory, and a relaxed, but rather complex, memory model [11]. In contrast to C99, handling concurrency and memory consistency mostly relies on the programmer. This means that the compiler cannot properly deduct from the source code how concurrency and memory consistency must be handled. For C++11, this means that a programmer has to define threads, and partition and balance the workload by hand. Moreover, every shared variable must be declared using `std::atomic<>`, and every access to it is subject to manual specification of the required memory orderings—C++11 defaults to non-shared variables, and strong consistency rules for shared ones. Because a compiler cannot determine concurrency and memory ordering properly by itself, these models are part of the programming model, as indicated in figure 3.2. Java 5.0 is similar to C++11, except that Java uses a slightly different model of computation: the abstract machine has a stack instead of registers, and it cannot change the instructions by itself[4]. Using shared variables is less complex than in C++11, but the relaxed memory orderings are still exposed to the programmer.

Go [93] requires the programmer to define so-called goroutines, which are lightweight microthreads. These threads can communicate via shared memory and a relaxed memory model, but it is preferred to use FIFO channels. Because the compiler understands the concept of channels—in contrast to concurrency in C99—it can handle them properly, which hides the underlying memory model from the programmer. Therefore, concurrency is under control of the programmer, but the memory model is not.

[4] Java's just-in-time compilation does modify the code, but this is part of the virtual machine and not of the language.



language (compiler)	C99 (`gcc`)	C++11 (`g++`)	Java 5.0 (`javac`)	Go (`gc`)	SANLP (`pn`+Espam)	Erlang (`compile`)	Haskell (GHC)
programming paradigm	imperative	object-oriented	object-oriented	imperative	sequential	functional	functional
model of computation	RASP machine	RASP machine	stack machine	register machine	register machine	λ-calculus	λ-calculus
concurrency model	single-threaded	threads	threads	microthreads, i.e. goroutines	KPN	Actor model	microthreads, i.e. sparks
memory model	irrelevant	relaxed	relaxed	relaxed, FIFO channels (preferred)	FIFO channels	FIFO channels	(originally) strict

Figure 3.2 – High-level overview of different programming models, indicating the models that are used in the compilation flow. The cross section of models that is exposed to the programmer, is indicated by the overlay.

The flow using pn [112] and Espam [87] takes a restricted form of a sequential input, namely SANLP with C functions, and extracts a process network, similar to a KPN. These processes communicate via channels. SANLP is a coordination language [71] that defines the relation between function inputs and outputs. These functions can be specified in any language, but as the default flow uses pure C functions, the model of computation is listed as a 'register machine' in figure 3.2 on the preceding page. Because the SANLP input is sequential, the concurrency appears to be hidden from the programmer. However, the sizes of the functions, which are connected by SANLP, determine the amount of concurrency. Therefore, it is still required that the programmer knows about concurrency and defines appropriate functions, in order to exploit parallelism of the architecture. So, a large part of the concurrency model is still included in the programming model. Nevertheless, among other details, synchronization and mapping is done automatically—this is an example of a programming model that does not expose the full concurrency model to the programmer, but only a subset of it.

Finally, both Erlang and Haskell are functional languages, based on λ-calculus. They handle concurrency differently. Erlang uses the Actor model, which requires explicitly defining separate processes that communicate via channels. For Haskell, GHC supports special functions to indicate that specific expressions could be done in parallel, but are not required to. Such an expression can still access all other expressions according normal scoping rules. So, only for Erlang, the concurrency model is included in the programming model, because the programmer defines separate computational units and their communication patterns, where Haskell only needs hints what could be done in parallel and what not.

Summarized, a programming model contains everything that has to be under control of the programmer, so it is the *programmer's view* on a programmable system. Features of the platform that are exposed by the programming model, cannot be handled automatically by the compiler or other tooling, but everything else can be done automatically and in a correct way. Figure 3.2 simplifies the programming models of the different languages, since the reality has more nuances; of course, Haskell still needs a bit of control regarding concurrency, and C++11 can do some memory consistency automatically. However, this overview captures the main idea of the languages. It also gives a characterization of programming languages in terms of the amount of abstraction and automated control of a platform.

3.5.2 Less is more

The smaller the overlap of the programming model and the underlying models of the platform is, the more tooling can do without guidance of the programmer, which in turn makes programming less error prone. So, the goal is to reduce it to just the model of computation, without having excessive overhead in lower layers. This is only possible when sufficient *knowledge* of the application is supplied to the compiler. In theory, when a compiler has all knowledge, then it can find the optimal translation. Otherwise, it makes either optimistic assumptions that could lead to incorrect behavior (e.g., all instructions of listing 3.1 on page 28 can be freely reordered), or conservative assumptions that could lead to overhead (e.g., take precautions such that all instructions are always executed in one specific order). Hence, it is essential to tell the compiler the intended behavior for correct and efficient code, which should be enforced by the programming model.

Although Go, pn+Espam, Erlang, and Haskell hide the lowest layer(s), it is not true that there is no control over them. It is possible to reformulate the source code of the application, such that a different compilation result is achieved. However, it is *impossible* for the programmer to make errors with memory model related issues, for example, and the compiler cannot break the application by apply optimizations related to these layers. This is in great contrast to C++11 and Java 5.0, which need to have guidance to compile the code in listing 3.1 correctly.

3.6 Platform and portability

In the end, all applications require hardware to run. Not only a processor is a necessity, also some form of memory, interconnect, and peripherals are essential. All hardware together is often labeled the platform. However, applications are rarely written on top of the 'bare metal' hardware; at least, an OS offers a more convenient environment for application programmers. Moreover, the OS and all available libraries possibly influence how the platform is used more than the hardware platform itself. Therefore, we use the following definition of a platform:

platform
The combination of the hardware and software that together shapes the environment for an application.

The figure on page 26 also includes the cross-layer 'platform' to indicate what is included when we speak of one. A desktop computer's platform might be characterized by {x86 + Linux + Pthread}, which indicates the models and implementation of all layers the platform consists of. An application only sees a platform through the programming model, which filters some aspects out and transforms others, as discussed in section 3.5.

Consider the code of listing 3.2 on the next page. For example, the desktop computer mentioned above can be programmed using C++11. Then, the compiler can map C++11's threads to Pthreads or Linux's native threads; writes to `X` can be translated to several instructions to store it atomically; and both writes are followed by an `mfence` instruction. This code is portable; when the program is compiled to {x86-64 + Windows}, threads are mapped to native Windows threads, and writes to `X` become just one instruction, which simplifies atomicity guarantees.

Compare the portability to the example of listing 3.3. Essentially, the same program is defined, but now in C99 using Pthreads. This is not portable, despite the initial P of the threading model. The C programming model allows using arbitrary libraries that exist on the platform, Pthread in this case. The Pthread library is not part of the

```
 1  #include <thread>
 2  #include <atomic>
 3  #include <iostream>
 4
 5  std::atomic<int> flag;
 6  std::atomic<double> X;
 7
 8  void thread1(){
 9      while(flag!=1){}
10      std::cout << X << std::endl;
11  }
12
13  void thread2(){
14      X = 0.42;
15      flag = 1;
16  }
17
18  int main(){
19      std::thread t1(thread1);
20      thread2();
21      t1.join();
22      return 0;
23  }
```

Listing 3.2 – Portable C++11 example

```
 1  #include <pthread.h>
 2  #include <stdio.h>
 3
 4  volatile int flag;
 5  volatile double X;
 6
 7  void* thread1(void* arg){
 8      while(flag!=1){}
 9      printf("%f\n",X);
10      return NULL;
11  }
12
13  void thread2(){
14      X = 0.42;
15      flag = 1;
16  }
17
18  int main(){
19      pthread_t t1;
20      pthread_create(
21          &t1,NULL,thread1,NULL);
22      thread2();
23      pthread_join(t1,NULL);
24  }
```

Listing 3.3 – Non-Portable C99 example

programming language, and compiling the application to {x86-64 + Windows} is not possible. A more subtle bug can be found when porting from {x86-64 + Linux + Pthread} to 32-bit hardware: writing X at line 14 is not atomic anymore, the output to the console can be garbled, and the compiler does not know that it has to take some effort to fix this. Hence, the application uses properties of the platform, which are outside of the programming model—another platform that implements the same programming model does not necessarily have identical properties. An application that is to be ported without any modifications to the source code should, therefore, only use properties that are defined by the programming model. A threaded C program is by definition not portable without additional effort.

As discussed in section 2.6, C is popular among all ten commercial systems. This is understandable, as most programmers know C, the language allows good control over the performance of the program, and a compiler can be built relatively easily, because C does not abstract much from the underlying architecture. However, it is very unlikely that any parallel application can be ported between all ten architectures, because concurrency within these platforms' programming models is as platform-dependent as inline assembly. The architectures, such as Epiphany-IV and OCTEON II CN6880, differ too much to offer the applications a common interface.

3.7 Related work

Most literature focuses on one of the models of the platform figure on page 26. To the best of our knowledge, no literature exists that integrates these models into one consistent platform view. This chapter organizes these existing models as layers of the platform, which the programming model exposes a specific cross section of. In turn, complexity is hidden from the programmer, and the compiler has enough knowledge to achieve high efficiency.

A common approach for high efficiency is to assume that the programmer supplies the task graph [51, 68, 87]. Then, tooling can find a mapping of this graph onto the hardware, given the constraints specified by the programmer. Although this approach hides low-level details of the hardware, it forces a programmer to conform to the given programming model for a specific multiprocessor architecture. This can give good performance, but still leaves the complexity regarding parallelization and partitioning, and the verification of it to the programmer. As we argued in section 3.5.2, it is better to reduce the programming complexity by proper abstraction layers. In chapter 5, we will do this for memory consistency, such that all consistency issues and portability are solved by the compiler, based on easy-to-formulate intentions of the programmer.

Another direction is to extend existing languages, like OpenMP and Cilk [18]. Then, existing programs in the base language are still compatible with the extensions and allow incremental development of the program. However, adding functionality to a language that was originally absent (like concurrency in C), still has the risk of not being analyzable [20, 71]. Moreover, optimizations that are oblivious to the extensions can break the program.

Lee [71] argues that extending is not a fruitful approach, and suggests using coordination languages. These languages have complementary features (such as concurrency) to the languages they coordinate, like UML for C++ programs. This helps to structure programming, because concurrency is made explicit, for example. However, no compiler has full knowledge of the complete application, as all of them focus on a specific language and feature set. In contrast, we target to increase the compiler's knowledge about the application, e.g., by annotations, that can lead to a highly efficient implementation, regardless the actual architecture—Chapters 5 and 6 make properties of the application more explicit, but reduce the programming complexity by abstraction layers at the same time.

Linderman et al. [73] propose the Merge framework, which consists of a coordination language that connects functions from a library. This library contains implementations for all different architectures. The framework offers a generic high-level description based on the map–reduce pattern. Again, using such a library makes the compiler oblivious to the intended behavior, which leads to a suboptimal solution, in principle.

Functional languages are attractive, because their model of computation naturally allows concurrency, and the abstractions hide low-level hardware complexities.

Since the relation between the source code and the compiled binary is not obvious, analysis of resource usage is usually difficult [50]. Hume [50] tries to close this gap, but is still only analyzable in the lowest language abstraction layers. Although functional programming is promising (as we will show in chapter 6), it is only slowly adopted, because the paradigm differs greatly from common programming practice.

A more holistic approach is presented by Jerraya et al. [62]. They define the programming model for a multiprocessor system as a matrix, consisting of API primitives for several abstraction layers, that is focused on hardware–software co-design. Their approach assumes a concurrent input description, based on message passing, and explicit communication and synchronization, and allows simulating the whole system with a certain speed–accuracy trade-off. In contrast, we focus on hiding complexity from the programmer (e.g., see chapter 5). Moreover, we investigate by means of the programming model how to deduct an efficient implementation from information-rich source code, where Jerraya et al. are more depending on run-time overhead by middleware layers, such as a hardware abstraction layer and resource management services.

3.8 Conclusion

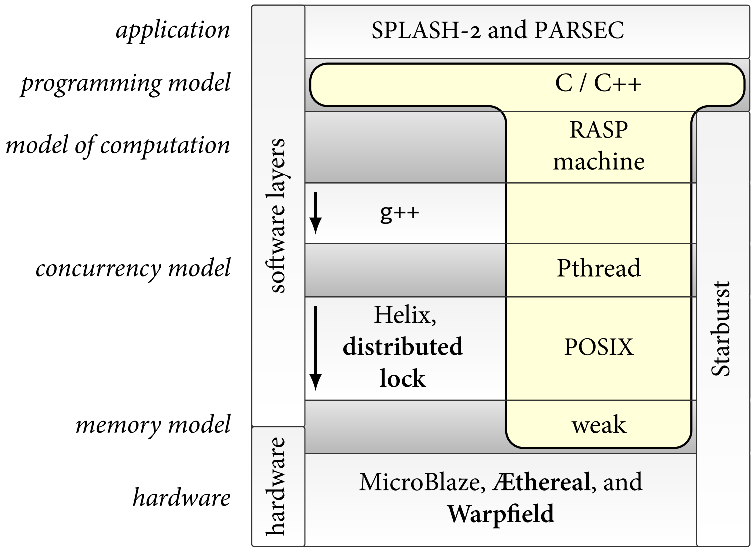
The programming model presents a platform in a specific, preferably elegant and convenient, way to the programmer. A programming model allows transparent implementation of certain details of the underlying platform, and therefore hides these details from the programmer. Which details are exposed to the programmer, differs per model, and hence per programming language.

An application is written using a specific programming model. In order to let the compiler or any other glue tool in the platform make optimal decisions about the implementation of the application, the intentions should be clearly expressed. This means that only and exactly the required dependencies for computation are defined. In C99 with Pthreads, for example, this is not the case; it cannot be determined from inspecting the source code whether two write operations depend on each other or can safely be reordered.

Finding a proper balance between expressiveness, freedom in implementation for the platform, and efficiency is hard. In general, a higher abstraction level allows a cleaner description of the hardware, which might be easier to program for. However, such an abstraction could come at a price. Subsequent chapters will show trade-offs between abstraction and costs.

This empty page leaves some room for random thoughts:

What is the point of reasoning about performance of a program, when that program still contains bugs? Can a C program really be considered efficient, if nobody knows whether it is bug-free?



Chapter 4 overview

4 Efficient Hardware Infrastructure

Abstract – *Experiments with threaded C applications in a many-core context show two issues: communication and synchronization costs. As bandwidth to memory is limited, core-to-core communication is favored. In a many-core setup, a connection-oriented NoC is often recommended. However, such a NoC uses hardware resources per connection, which is quadratic in the number of cores. This chapter presents the Warpfield NoC, which scales linearly and improves application performance, as it is work-conserving, but has a higher worst-case latency bound. Moreover, a distributed lock algorithm is proposed to implement mutexes without polling memory. For the given programming model, the resulting system scales close to linear in terms of performance.*

In chapter 1, we have seen that future embedded processors are likely to be many-core systems. Existing commercial processors all try to scale the number of cores, but maintain the threaded C programming model on top of a shared-memory architecture as much as possible, which is discussed in section 2.6. Moreover, these many-core systems have in common that the memory bandwidth per core is reduced, which increases the importance of locality. Therefore, expensive off-chip memory communication is avoided by using distributed and non-uniform memory, such as caches and SPMs, and direct core-to-core communication is preferred.

Because a traditional bus is unsuitable as infrastructure for such a many-core architecture, NoCs have been developed. Among other properties, a NoC can be classified as *connection-oriented* or *connectionless* [16, 78]. The former defines that the NoC's hardware has knowledge of a connection between two communicating

Large parts of this chapter have been published in [JHR:3, 4].

entities, like two processes or a process and a memory. As such, the hardware allocates resources, such as buffer space and bandwidth capacity, for a specific channel. A typical example is a circuit-switched network. A connectionless network does not know about connections, and transfers data in chunks as being issued to the network. Packet-switched networks—without virtual channels—are typically connectionless. When the network hardware is connectionless, a connection-oriented protocol can still be used on top of it. At a different scale, using TCP over IP in a LAN is one example of that.

To give real-time guarantees, the use of a connection-oriented guaranteed-service NoC is proposed [16, 37, 47, 114]. This allows performance analysis of applications in isolation, because communication of one application can only have a bounded influence on other running applications. However, for shared memory, a connection-oriented NoC requires connections between every core and (local) memory. This is expensive because a hardware cost is associated with every connection. A connectionless NoC is less expensive, but the performance per connection is uncertain.

An additional effect of a connection-oriented NoC is that (FIFO) channels impose a specific form of synchronization: one-to-one, where both participants of the synchronization are known upfront. This is in contrast to conventional usage of mutexes, of which it is unknown which thread is going to lock it next. A (distributed) implementation of a mutex on top of a NoC architecture is not trivial.

This chapter will discuss two trade-offs in the implementation of many-core hardware, and Starburst specifically, given the threaded C programming model: the architecture of the interconnect; and the realization of synchronization on top of this interconnect. Both trade-offs are fully transparent to the application, because they exist outside of the programming model. The overview figure of this chapter on page 44 depicts this in some more detail: we use SPLASH-2 and PARSEC applications written in C using Pthreads, on top of a POSIX-like OS and a weak memory model.

On the interconnect side, Æthereal is replaced by our Warpfield interconnect. Experimental evidence is provided that confirms that substitution of a connection-oriented NoC by a connectionless one in a real-time DSM system reduces hardware costs significantly. Furthermore, it improves the processor utilization, but *does* compromise the analytically computed worst-case behavior. However, an increase in the uncertainty introduced by the connectionless interconnect is *not* confirmed by the experimental results.

One generic usage of core-to-core communication is synchronization. Synchronization is commonly implemented using atomic read–modify–write *(RMW)* operations, which poll main memory. As this is a scarce resource, and chapter 2 shows the trend that architectures shift away from atomicity in memory operations, such a polling-based method should be avoided. To bypass shared memory, we propose an efficient distributed lock algorithm that implements the Pthread mutex. We show that using a low-cost inter-processor communication ring for synchronization reduces the required SDRAM memory bandwidth. Additionally, the distributed lock reduces the average latency of locking, by exploiting the locality of mutexes. As a result, the throughput and execution time of applications improve.

4.1 Communication patterns and topology

Before we discuss the experimental setup in the next section, we first look more closely to the requirements that applications impose on the interconnect. Current systems allow chaotic and undisciplined use of shared memory [4], and threaded programs make use of this property. Naively, threading therefore requires the hardware to implement all-to-all communication between the cores. On the other extreme, streaming applications, like multimedia applications, can often be described in a KPN model. In contrast to threads, such a model clearly defines the communication pattern of the application. Because such a KPN is static for a significant amount of running time of the application, the hardware (or NoC configuration) can be tailored toward this pattern. However, fixating the NoC topology for only one application is not feasible for a general-purpose platform. We differentiate the following general types of communication streams for a typical DSM architecture:

1. *Memory-to-core for instructions:* Since all cores run code from main memory, the instruction cache must be filled regularly.
2. *Core-to-memory and memory-to-core for local data:* All data that is local to a core, such as kernel data, (most of the) process stack, and specific data on the heap, can safely be cached. The data cache has to flush and fill cache lines regularly.
3. *Core-to-memory for shared data:* In a shared-memory setup, the main memory is used to communicate data between cores. Writing shared data to memory effectively means that this data is to be sent to another core. This data can be cached, but requires a coherency protocol then to give guarantees how these writes are observed.
4. *Memory-to-core for shared data:* As being the counterpart of the core-to-memory stream, shared data that is sent to a specific core, is read by that core.
5. *Core-to-memory for synchronization:* Similar to writing shared data, writes regarding synchronization are intended to be sent to another core.
6. *Memory-to-core for synchronization:* For synchronization in a shared-memory system, synchronization data structures in main memory are polled.
7. *Core-to-core for shared data:* Shared data that is written into other core's SPM directly. This bypasses main memory.
8. *Core-to-core for synchronization:* Synchronization data structures are small, because they do not hold data itself, except for the internal state. Therefore, they can also be implemented over the SPMs.

Of these types of communication streams, an increased read latency for streams 1 and 2 directly has impact on the performance of the application. During a cache miss of either the instruction or data cache, the core stalls until the required data is fetched from background memory. As every memory model prescribes that writes should be visible to the executing process immediately (see also section 5.1.1), an increased latency of the writes of stream 2 impact the performance too, because a successive read has to stall on it. Therefore, these streams are considered to be *latency-critical*. In fact, every read from main memory is latency-critical, so streams 4 and 6 are classified similarly.

Streams 7 and 8 can affect the performance of the application, when the receiving process must wait for its data. Usually, the performance of any process is not precisely known. For applications without a strict feedback loop, an approach to tolerate differences in data production rates is to apply buffers in the channels between processes. These buffers can contain posted writes, and allow pipeline concurrency, which might improve the throughput. Since an (incidentally) increased latency in these channels is compensated by buffers, these streams are *latency-tolerant*. Similar to the core-to-core streams, the streams 3 and 5 are latency-tolerant. Because accesses to main memory should be avoided, as this is a scarce resource, these streams should be avoided in favor of the core-to-core alternatives.

Two different types of interconnects are required to accommodate the different types of streams above: a many-to-one latency-critical core-to-memory-to-core interconnect, and a many-to-many latency-tolerant core-to-core interconnect. The hardware topology must always allow core-to-memory-to-core communication, because this channel is required to execute programs. The core-to-core hardware topology is more flexible, because it is unlikely that all cores communicate to all cores simultaneously. A practical implementation could allow that only a (configurable) subset of all cores is accessible at the same time. However, limiting the communication pattern of an application complicates programming; the threading model assumes that all threads and cores are always accessible, so threading does not match these limitations of the NoC.

4.2 Baseline: Starburst with Æthereal

We use Starburst with the Æthereal NoC as a baseline. This section describes the system, the hardware requirements, and core utilization measurements. The experiments show several shortcomings, which are addressed in subsequent sections.

4.2.1 8-core setup

The system is organized as a tiled architecture, where every tile contains exactly one core, as discussed in section 2.7. The Æthereal [47] interconnect is a connection-oriented NoC. The NoC contains a mesh of routers. Every router has a routing table, which defines which input port should be connected to which output port every clock cycle, i.e. every *slot*. The router continuously and repeatedly steps through this table, which results in TDM arbitration of incoming packets. Wormhole connections are allocated through the mesh by defining the routing tables, such that

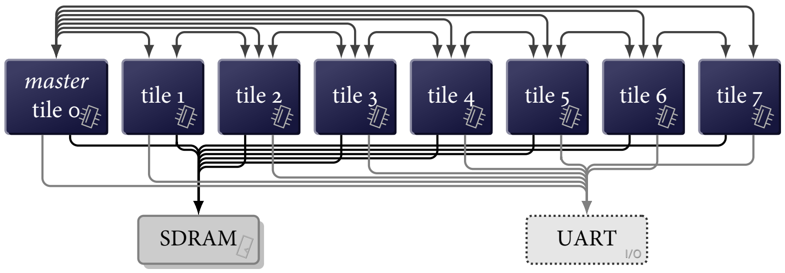


FIGURE 4.1 – Æthereal NoC topology for Starburst

flits, which are inserted in the network at a specific moment in time, are contention-free routed to their destination. Network interfaces are connected to the mesh. They contain buffers for both ends of every connection, and control when flits are injected into the mesh. The combination of buffer size and TDM table contents determines the route, bandwidth, latency, and jitter per channel.

The number of routers, network interfaces, and buffers is determined at design time. This defines the maximum capacity of the network. Therefore, the better the communication pattern of the application (domain) is known, the better the network dimensions can be chosen. At run time, a specific configuration can be chosen, such that the configured topology matches the communication pattern of the application. From application's perspective, channels do not interfere and offer a point-to-point connection between master and slave, e.g., core and memory.

In conformance to section 4.1, the NoC should support concurrent channels from all cores to memory, and as many channels between tiles as possible. Although our Virtex-6 LX240-T FPGA has enough resources to accommodate 32 MicroBlazes, synthesis shows that an 8-core design with a fully connected Æthereal configuration does not fit in the FPGA. As a—naive and non-generic, but simple—solution, Æthereal is configured such that every MicroBlaze can communicate with 1) the main memory, 2) one 'master' MicroBlaze that manages startup of the other cores and application, 3) one peripheral for UART output, and 4) both neighbors for (limited) core-to-core communication. This topology is shown in figure 4.1. By default, Starburst includes a tile reserved for Linux and several peripherals. This tile and all of its peripherals are omitted to save resources, as it is not used in the experiments.

Æthereal is configured with very low bandwidth requirements to maximize configuration freedom[1], and the buffer sizes per channel in the network interfaces are set to contain one burst of one cache line of the MicroBlaze's cache, which is 32 byte. This configuration fits in the FPGA, and will be used as reference design.

[1]In fact, when realistic bandwidth requirements are set, a suitable configuration cannot be found. Forcing a different internal network structure or choosing the number of input/output ports differently does not help.

Table 4.1 – Virtex-6 LX240-T FPGA resource usage of system with Æthereal

	LUTs		FFs		BRAMs
master MicroBlaze	2 664	(3.2 %)	2 239	(2.6 %)	8
master tile[a]	2 372	(2.9 %)	2 385	(2.8 %)	9
7× slave MicroBlaze	2 461	(3.0 %)	2 003	(2.3 %)	8
7× slave tile[a]	1 184	(1.5 %)	714	(0.8 %)	5
interconnect	46 535	(57.0 %)	56 274	(65.8 %)	0
peripherals	4 542	(5.6 %)	5 594	(6.5 %)	10
total	81 628	(100.0 %)	85 511	(100.0 %)	118

[a] The tile includes local memories, a timer, PLB and bridges (see figure 2.1 on page 15).

4.2.2 Synthesis results: exponential costs

The synthesis results of the 8-core reference design for a Xilinx Virtex-6 LX240-T FPGA at 100 MHz is shown in table 4.1. The table shows that the master MicroBlaze is slightly bigger than the slaves are, which is caused by additional debug and performance measuring support. This support does not influence the performance of the core. In the table, the resources required for the memory controller are included within the resources of the peripherals.

Still, the interconnect is the biggest part of the system. The NoC contains 1.5 times more lookup tables *(LUTs)* and 2.4 times more flip-flops *(FFs)* than all MicroBlaze tiles together. Practical reasons for that result are that MicroBlazes are small and optimized for FPGAs, and Æthereal does not map to an FPGA well.

However, there is a more fundamental problem: every connection in a connection-oriented NoC has associated hardware costs. In case of Æthereal, most of the area is used by buffers that are necessary to guarantee throughput. This corresponds to the findings of Goossens and Hansson [47]. When we want to have a fully connected interconnect for all core-to-core channels, a connection-oriented network becomes expensive since a quadratic number of buffers is required.

The trend of scaling to many cores is demonstrated by figure 4.2. The figure shows synthesis results for systems with up to 16 cores and a fully connected Æthereal. The points in the graph indicate resource usage after synthesis of the interconnect alone, the lines indicate the (calculated) size of the cores and peripherals, as of table 4.1. No bandwidth and latency requirements are applied and all settings are kept the same, except for the number of routers inside the interconnect—they had to be increased to accommodate the increasing number of links, resulting in the discontinuities. The precise resource utilization depends on many settings, but the trend is clear: the figure shows a superlinear growth in resources. In fact, when having 13 cores or more, the interconnect alone does not fit in our FPGA anymore. For ASIC synthesis, the ratio between hardware costs of the cores and network will be different, because the hardware description is mapped to other technology

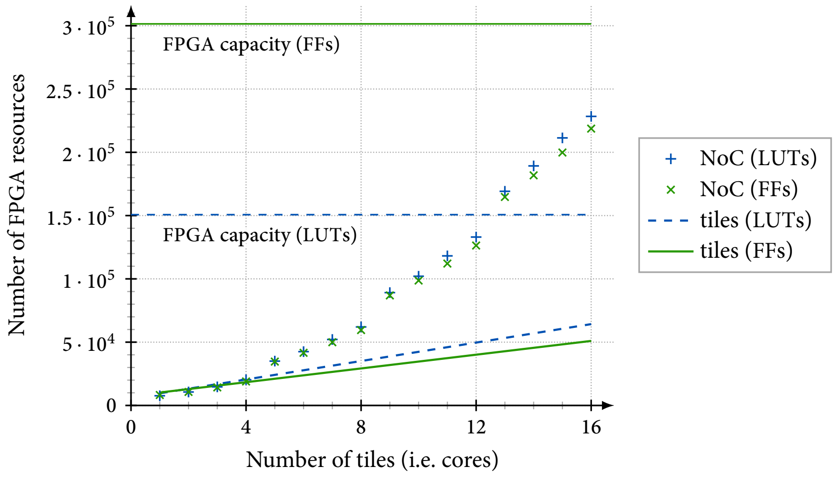


Figure 4.2 – FPGA resource usage with a fully connected Æthereal NoC

primitives. However, the trend is the same; the network will outgrow the size of the tiles when scaling to tens or hundreds of cores.

Although the hardware costs are already pushing the practical limits, it is as least as important how applications perform on the hardware.

4.2.3 Core utilization by benchmark applications

To analyze the performance of the platform, we ported the SPLASH-2 `radiosity`, `raytrace`, and `volrend` applications (see section 2.8.1). Using the trace port of the MicroBlaze, microarchitectural data is gathered about what the core is doing every clock cycle. We measured these statistics during the main, parallel application loop of each of the applications. It is safe to assume that all cores exhibit the same behavior, because all applications are designed such that the workload among all cores is balanced and equivalent. Table 4.2 on the next page shows the distribution of clock cycles over the five most important states a core spends time on. These states are:

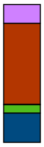
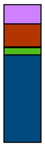
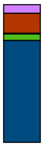
» *execution*, which indicates that the core executes instructions in a normal fashion, and gets all instructions and data directly from the cache, but it includes stalls due to branches and register hazards;

» *I-cache miss* labels the stalls because of instruction cache misses;

» *read data* is the time the core stalls on uncached data reads and data cache misses;

» *write data* captures all stalls on uncached writes and stalls due to data cache flushes; and

Table 4.2 – MicroBlaze utilization

event	`radiosity`		`raytrace`		`volrend`	
🟪 I-cache miss	13.7 %		14.0 %		6.3 %	
🟥 read data	58.7 %		16.7 %		14.4 %	
🟨 write data	0.3 %		0.5 %		0.6 %	
🟩 other	6.1 %		5.3 %		4.7 %	
🟦 execution	21.2 %		63.4 %		73.9 %	

» *other* includes overlapping and indecisive events, such as a simultaneous instruction and data cache miss.

It turns out that even with a high instruction cache hit rate—99.1 %, 99.7 %, and 99.9 % for `radiosity`, `raytrace`, and `volrend`, respectively—handling the instruction cache misses takes a significant amount of time. The performance is limited by the high memory read latency: a read takes 77 clock cycles on average, where 15 cycles are spent in the SDRAM controller to process a read, and the rest in the NoC to traverse it twice. Additionally, table 4.2 shows that for `radiosity`, the stall time on data reads is high. This is mainly caused by data structures that are placed in uncached memory, because no cache coherency is available. Therefore, the network latency greatly influences the performance.

Traversing the NoC is expensive, because: 1) one memory request packet waits for multiple TDM slots (which are non-contiguous) in the network routers, even when the NoC is idle; and 2) the response also has to wait for its slots, because the arbitration of the request and response packets are unrelated, even though the memory controller has a relatively predictable average response time.

4.2.4 Shortcomings of connection orientation

In a DSM architecture that is programmed using a threaded C approach, where the communication pattern of the application is not restricted, a many-to-many (or all-to-all) network topology is required. A connection-oriented network that supports this pattern, has to allocate hardware resources per channel. Regardless of the efficiency of the network, the hardware scales quadratic to the number of cores in the system, and therefore always becomes a dominant factor in hardware design.

Moreover, the experiments show that Æthereal does not perform well for latency-critical traffic to main memory. This results from the fact that the network is composable, where channels are designed not to influence each other, i.e. always perform as in the worst-case scenario. As a result, the latency of the network is relatively high, even when other channels are idle. Cache misses, which generally occur at unpredictable moments, do not benefit from guaranteed-bandwidth channels. This also holds for accessing shared data in a shared-memory model, which C is based on; the uncertainty of the performance in execution of a threaded program is too high to justify using hardware bandwidth guarantees. Therefore, it is sufficient to have an interconnect that performs well on average, but allows (bounded) bandwidth interference of different channels.

The interconnects of the architectures described in section 2.4 are mostly focused on cache coherency traffic. They do not give guarantees about bandwidth or latency. As all architectures are application agnostic, they can therefore be classified as connectionless. For none of the systems, the NoC's influence on the real-time behavior is known. Next, we will present an interconnect that is also connectionless, but predictable.

4.3 Warpfield: a connectionless NoC

The previous section identified two problems of connection-oriented NoCs: super-linear scaling of hardware resources and high latency for memory reads. A connectionless NoC is likely to scale better than a connection-oriented NoC, because hardware resources are used per processor instead of per connection. The most important reason to use a connectionless NoC in a DSM architecture instead, is that it naturally scales linearly to the number of processors. We designed Warpfield, which is a connectionless NoC, and differentiates between all-to-all latency-tolerant and all-to-memory latency-critical traffic, which is similar to the separation in figure 2.2 on page 19.

4.3.1 Bitopological architecture

Figure 4.3 on the following page depicts the implementation details of the new interconnect. We chose a ring for the latency-tolerant traffic, which corresponds to the interconnect in the upper part of the figure, because of its simplicity. The ring is built of as many chained identical segments as there are tiles in the system. Every segment allows one core to access the ring. Every core can write a data word to an address that matches a local memory of any tile. This address–data pair is put in a small FIFO, awaiting injection into the ring. Packets already on the ring have priority over those that are waiting in the FIFO. Because the local memories in the tile always accept packets, and the ring itself does not block, it is sufficient to have one set of registers that connects two segments. So, a packet traversing the ring hops one tile per clock cycle towards its destination.

There is no traffic shaping for the ring; so if, for example, core 1 writes every clock cycle to core 0, no other cores would be able to access the ring anymore[2]. However, in practice, this is not a limitation; in contrast to hardware accelerators, MicroBlazes are not fast enough to generate, send and receive data at such high rates. Moreover, our benchmark applications do not use the ring in this way, because the local memories are too small for most data structures.

[2]Dekens et al. [39] extended the ring such that it does give per-processor bandwidth guarantees.

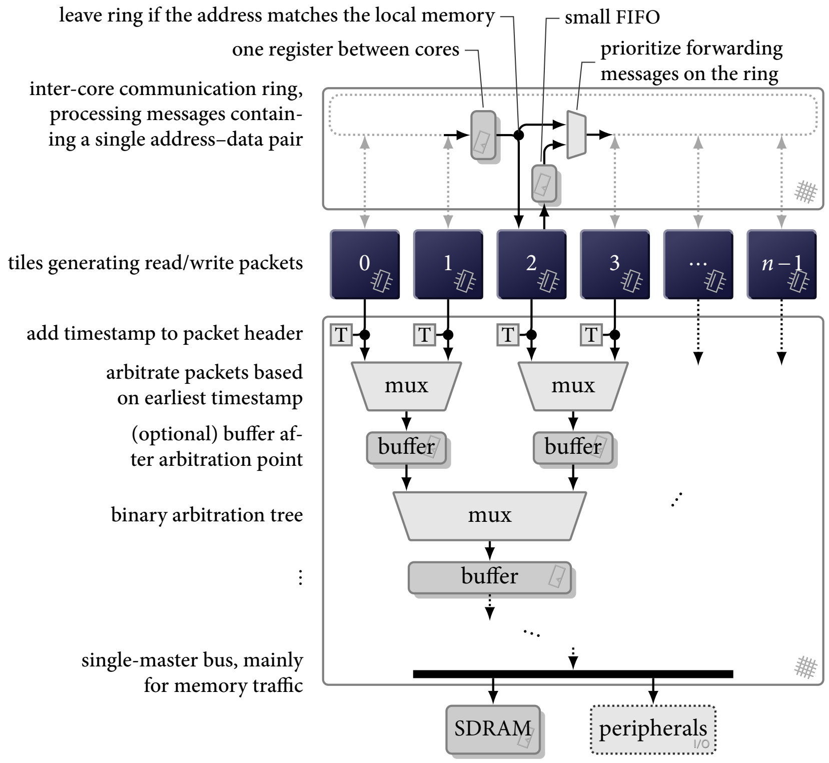

leave ring if the address matches the local memory
small FIFO
one register between cores
prioritize forwarding messages on the ring
inter-core communication ring, processing messages containing a single address–data pair
tiles generating read/write packets
0
1
2
3
…
n−1
add timestamp to packet header
T
arbitrate packets based on earliest timestamp
mux
(optional) buffer after arbitration point
buffer
binary arbitration tree
single-master bus, mainly for memory traffic
SDRAM
peripherals
I/O

Figure 4.3 – Structure of a system with Warpfield

In contrast to the ring, the latency-critical part, which connects the cores to the memory and peripherals, has to handle a lot of traffic. This interconnect must adhere to the following requirements:

1. starvation-free scheduling, such that liveness of all cores is guaranteed;
2. work conserving to optimize for average read latency;
3. scale linearly in hardware costs to the number of cores; and
4. pipelined and decentralized arbitration to avoid long wires for high performance.

A traditional bus cannot satisfy the last requirement. We implemented a tree-shaped network with first-come-first-served *(FCFS)* arbitration that conforms to all requirements. This arbitration network is designed such that it bridges the physical distance on a chip, without decreasing the maximum clock frequency. The bottom part of figure 4.3 shows the structure of this new interconnect, having arbitration of *n* cores to a memory controller and peripherals. When desired, multiple arbitration trees can be instantiated for higher bandwidths. However, as the off-chip memory interface is the bottleneck in the system anyway, we only use one tree. The network supports read and write requests, which are issued by the cores. Every read and write request of a processor is packetized, containing one command, one address, and multiple data flits.

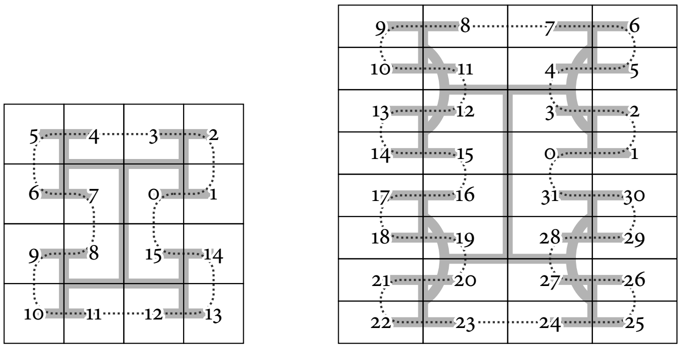
Following the path from the core to the memory in figure 4.3, the arbitration tree works as follows. A packet gets a timestamp and processor ID upon injection, which is sent along with the packet. The timestamp can be generated locally to every core, as long as the timestamp generators are synchronized, e.g., during reset. The FCFS is fair when all generators are in sync. When a generator falls behind, a later packet will receive a lower, i.e. earlier, timestamp than packets that get timestamps of properly running generators. In this case, the network still works, but these packets get slightly prioritized, which in turn influence the maximum latency of other packets. Therefore, a trade-off can be made between proper synchronization of the generators, which might be hard to realize in hardware, and the accuracy of control over the latency and priority of packets. Drifting clocks, however, will lead to large differences in timestamps, after which starvation-free arbitration cannot be guaranteed anymore. The number of bits for the timestamp depends on how much time there can be between two packets that are in the tree simultaneously. To determine this period, one has to take the worst-case waiting time for a packet into account.

Next, the packets are sent through a binary arbitration tree that multiplexes n processors to one bus master, where every step in the tree is a multiplexer that does local arbitration of two inputs. This arbitration point lets the packet with the lowest timestamp precede. Rearbitration is only done between packets. After every mux, a small buffer can be placed for shorter wires or left out for lower latency. Therefore, multiple packets can be 'in flight' towards the root of the tree. By means of backpressure, requests can be stalled by subsequent muxs and the bus slave.

At the root of the tree, figure 4.3 refers to a 'bus'. However, this is essentially just a demultiplexer from the arbitration network to the memory and peripherals. In contrast to a traditional bus, the bus itself has only one master, and can be kept physically close to the bus slaves. Finally, the response packet will be sent back via a similar tree, but demultiplexes one to n cores, based on the processor ID (which is not shown in the figure for simplicity). Because there cannot be contention in the response tree—the MicroBlaze awaits the response and will always accepted the data immediately—arbitration is not required as well as a backpressure mechanism.

As discussed above, the network is a (balanced) binary tree. Therefore, the total number of multiplexers in the network equals $n - 1$. Hence, the hardware requirement scales linear to the number of processors.

The distance between core $n - 1$ and core 0 seems to be large, as visualized by figure 4.3. Between their two ring segments, there is only one register. Therefore, the cores are likely to be placed closer together during floorplanning of the hardware design. However, Warpfield uses two different interconnection topologies, which might seem to have conflicting constraints on the placement.



(a) 16 cores

(b) 32 cores

Figure 4.4 – Possible 2D floorplans of Warpfield's combined ring and tree topologies

In practice, this combination of topologies is realizable, as depicted in figure 4.4. Figure 4.4(a) shows a 2D layout of 16 cores, where the dotted line indicates the ring, and the H-tree the arbitration network. We assume that the memory controller can be squeezed in at the center of the design—cores do not necessarily have to be rectangular shaped. Although not as symmetrical, figure 4.4(b) shows a similar layout for 32 cores. The precise layout depends on many aspects, such as the position of memory banks in the FPGA and the pinout of the memory module. However, it shows that the networks are reconcilable.

4.3.2 Improvements in hardware and software

The synthesis results of an 8-core system with Warpfield, as depicted in figure 4.3 on page 54, for the Virtex-6 LX240-T FPGA at 100 MHz is shown in table 4.3. The table shows that the MicroBlaze tiles and peripherals slightly differ from table 4.1 on page 50 because the NoC interface changed. The total system uses about half the resources of the one with Æthereal. The interconnect itself is significantly smaller, mainly because fewer buffers are used and the buffers are smaller. Additionally, because packets cannot be interrupted, which is the case with non-contiguous TDM slots in Æthereal, a request can be processed immediately upon arrival. As a result, the receiving logic becomes simpler. With such resource usage, even a 32-core system fits in the FPGA. However, for fair comparison of the software performance, an 8-core system is used for all comparisons.

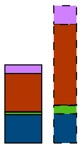
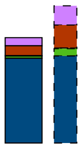
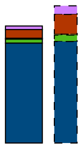
Not only the hardware costs, but also the performance of the benchmark applications improved. Table 4.4 compares the utilization of both experiments. It shows the measured time the cores spend in every state, relative to the measurements on the reference design (see table 4.2 on page 52); bars to visualize the measurements; and equivalent bars of the experiments with Æthereal (dashed).

Table 4.3 – Virtex-6 LX240-T FPGA resource usage of system with Warpfield

	LUTs		FFs		BRAMs
master MicroBlaze	2 664	(6.6 %)	2 239	(6.9 %)	8
master tile[a]	2 613	(6.5 %)	2 680	(8.2 %)	5
7× slave MicroBlaze	2 461	(6.1 %)	2 003	(6.2 %)	8
7× slave tile[a]	1 184	(2.9 %)	715	(2.2 %)	5
interconnect[b]	4 603	(11.5 %)	2 750	(8.5 %)	0
peripherals	4 754	(11.8 %)	5 832	(17.9 %)	10
total	40 149	(100.0 %)	32 527	(100.0 %)	114

[a] The tile includes local memories, a timer, PLB and bridges (see figure 2.1 on page 15).
[b] Main different with respect to table 4.1 on page 50.

Table 4.4 – MicroBlaze utilization with Warpfield

event	`radiosity` [a]		`raytrace` [a]		`volrend` [a]	
🟪 I-cache miss	6.3 %		5.9 %		2.6 %	
🟥 read data	27.9 %		7.1 %		6.4 %	
🟨 write data	0.2 %		0.4 %		0.6 %	
🟩 other	1.9 %		1.6 %		2.9 %	
🟦 execution	20.6 %		61.7 %		72.7 %	

[a] Right (dashed) bars indicate the performance in the reference design with Æthereal (see table 4.2 on page 52).

The total execution time is reduced for all three applications to 56.9 %, 76.6 %, and 85.2 %, with respect to the total execution time of the experiments on the reference design. Moreover, the time the processor stalls at the instruction cache misses and reads from the memory is roughly halved. This result can be contributed to the reduction in the read latency of the memory, which is now 37 cycles on average under full load and 25 cycles when idle (where the memory controller still consumes 15 cycles). The time spent in execution did hardly change, because the processor's speed did not change. The utilization—the ratio of *execution* to the total time—of the processor can easily be calculated based on these results. It improved significantly from 0.21 (`radiosity`), 0.63 (`raytrace`), and 0.74 (`volrend`), to 0.36, 0.80, and 0.85.

4.3.3 Bounded temporal behavior

Although the previous section shows that the average performance increased after replacing Æthereal by our connectionless network, this does not give guarantees for real-time behavior. Additionally, FCFS, which is used in the arbitration tree, is not known to be fair in general and can be outperformed by other schedulers [120].

However, this section will prove that the tree with FCFS can be used in a predictable system. The key aspect that FCFS can be used, stems from the fact that when the tree cannot accept more packets, the MicroBlaze stalls. Therefore, the number of packets that can be injected within a period of time, is limited.

We look only at the arbitration from n initiators (MicroBlazes) to one target (the root of the tree, which is connected to the memory and peripherals), assuming that the target always can process requests, and there is enough bandwidth from the target back to the initiator. Recall, the connection type that is serviced, is latency-critical. Therefore, packets do not have a deadline, but should be handled 'as soon as possible'—but starvation-free. Packets are non-preemptive, and flits of the same packet are always contiguous when they are injected in the network.

We define the *service time* $S(q)$ of a packet with type q and length ℓ_q, which is the time duration a packet spends in total in the network. This is the total amount of time between a packet arrives at a leaf of the tree and gets its timestamp, and the moment the last flit leaves the root and therefore is serviced. Hence, S depends on the total latency introduced by the network and the length of the packet. An initiator can only inject a new packet when the last flit of the previous one is injected in the tree.

In case of the best-case service time, denoted S_{bc}, a packet is not hindered by other packets, and is only hold up by the buffers after a multiplexer in the tree. Given a balanced binary tree, a packet therefore encounters $\lceil \log_2 n \rceil$ buffers in its path. A multiplexer in the (binary) arbitration tree can only process one flit per time unit, i.e. clock cycle. Processing the first flit of a packet in a tree without buffers is done by just combinatorial logic, which takes zero clock cycles; the other flits will follow in the subsequent clock cycles. Traversing a buffer always takes at least one time unit, even if the buffer is empty. Therefore, the best-case service time of a packet of type q is

$$S_{\text{bc}}(q) = \ell_q - 1 + \lceil \log_2 n \rceil. \tag{4.1}$$

The worst-case service time, denoted S_{wc}, is S_{bc} plus the time that the packet is obstructed by all flits in (the buffers of) the tree and the largest packet possible just being injected by all other $n-1$ initiators. Given $n-1$ multiplexers in the tree, the total buffer capacity is $(n-1)\beta$, where β denotes the buffer capacity in the number of flits after a multiplexer. However, only the flits that cross the path of the packet and therefore compete for arbitration have to be taken into account. So, the $\lceil \log_2 n \rceil$ flits in between the packet concerned and the root of the tree will not obstruct the packet, as they travel with the same speed towards the root and do not influence the arbitration of the packet. Hence, given a packet of type q, the worst-case service time is

$$\begin{aligned} S_{\text{wc}}(q) &= S_{\text{bc}}(q) + (n-1)\beta + (n-1)\max_{q' \in Q} \ell_{q'} - \lceil \log_2 n \rceil \\ &= \ell_q + (n-1)\left(\beta + \max_{q' \in Q} \ell_{q'}\right) - 1, \end{aligned} \tag{4.2}$$

TABLE 4.5 – Measured average packet issue intervals ψ, aggregated for all 8 cores (in clock cycles)

packet type in Q	ℓ	`radiosity`	`raytrace`	`volrend`
read word[a]	2	10.4	47.6	60.4
read burst[b]	2	40.7	51.4	100.3
write word[c]	3	763.2	6 197.0	4 187.5
write burst[d]	10	18 078.9	7 875.2	10 049.1

[a] Causes: uncached data read
[b] Causes: instruction cache miss; data cache miss
[c] Causes: uncached data write; data cache word flush
[d] Causes: data cache line flush

where Q denotes the set of all possible types of packets. Hence, the service time is bounded, and therefore the arbitration is starvation free.

As shown above, the worst case depends mostly on the size of the largest packet. In our system, there are four types of packets: a word and burst read request, both consisting of two flits (command and address); a write request of three flits (also includes data); and a burst write of ten flits (having eight data flits). For the 8-core Warpfield system, where $\beta = 2$ flits, the worst-case service time of a read packet can be calculated as $S_{\text{wc}}(\text{read}) = 85$ cycles, where $S_{\text{bc}}(\text{read}) = 4$ cycles. In contrast, the worst-case service time in the reference system with Æthereal (as of section 4.2) is $S_{\text{wc}} = 84$ cycles. However, Warpfield's *measured* total read latency, or round-trip time of a read request, is 30.66 cycles on average for all four applications. This measurement includes the 15 cycles required by the SDRAM controller, and the latency of the return packet, which is also $\lceil \log_2 n \rceil$. So, the measured average-case service time is about 12.66 cycles.

S_{wc} for our interconnect is high, because a packet must wait for every core issuing the largest packet simultaneously. However, this is a very unlikely situation, as the largest packets are burst writes, and these are rare. The measured average interval ψ between packets is listed in table 4.5. The table shows that for all measured applications, a burst write is issued two to three orders of magnitude less than a read.

The fact that burst writes are rare, explains why the measured performance in section 4.3.2 improved, although the worst-case service time increased. Namely, burst writes *can* interfere with read requests, but there are not enough burst writes to interfere with them *all*—which is assumed for S_{wc}. Let us calculate the expected interference, given the measured packet issue intervals.

Per time period τ, the number of packets of type $q \in Q$ issued by one of the n cores, denoted $I(q, \tau)$, can be calculated as

$$I(q,\tau) = \frac{1}{n} \cdot \frac{\tau}{\psi_q}. \tag{4.3}$$

For example, within a time period τ, `raytrace` issues per core on average

$$\begin{aligned} I(\text{read}, \tau) &= I(\text{read word}, \tau) + I(\text{read burst}, \tau) \\ &= \frac{1}{8}\left(\frac{\tau}{47.6} + \frac{\tau}{51.4}\right) \qquad \text{(for raytrace)} \end{aligned}$$

read packets. Packets can be hindered by flits sent by the other $n-1$ cores. Hence, the number of these flits, denoted $H(\tau)$, equals

$$H(\tau) = \frac{n-1}{n} \sum_{q \in Q} \ell_q \frac{\tau}{\psi_q}. \tag{4.4}$$

For `raytrace`, the number of interfering packets per time period τ is

$$H(\tau) = \frac{7}{8}\left(\frac{2\tau}{47.6} + \frac{2\tau}{51.4} + \frac{3\tau}{6197.0} + \frac{10\tau}{7875.2}\right). \qquad \text{(for raytrace)}$$

So, every read packet waits on average at most for

$$\frac{H(\tau)}{I(\text{read}, \tau)} \approx 14.3 \qquad \text{(for raytrace)}$$

interfering flits, and therefore 14.3 clock cycles. Including the latency of the memory controller and traversing the tree back to the core, the average latency of a read request under full load of all applications can be calculated as 37.86 clock cycles, which is close to what we measured.

Whether averaging interfering flits over multiple requests is allowed or not, depends on the real-time requirements of the platform. This section showed the trade-off between a cheap (in terms of hardware costs), and average-case high-performance interconnect, versus a more robust (in terms of temporal behavior), but expensive interconnect. As Warpfield fits our needs better, consecutive sections will continue to use this interconnect.

4.4 Inter-core synchronization profile

The previous section focused on the communication between cores and memory. The inter-core interconnect was not used for application data, mostly because of limited memory size. As synchronization in principle does not need large data structures, this is a possible use of the ring.

In modern general-purpose chips, synchronization is usually polling-based using atomic read–modify–write *(RMW)* operations as building blocks [35, 79], which are either hidden in a lock library or used by the programmer directly. RMW operations have a relatively low latency and are wait-free [53]. However, they require a cache-coherent system, which is hard to realize in general and absent in the Intel SCC,

Table 4.6 – Applications for synchronization experiments

benchmark set	application	mutexes
PARSEC	`fluidanimate`	4 403
SPLASH-2	`radiosity`	26 034
	`raytrace`	35
	`volrend`	37

for example. Without cache coherency, RMW operations induce traffic to external SDRAM, which we try to relieve as much as possible.

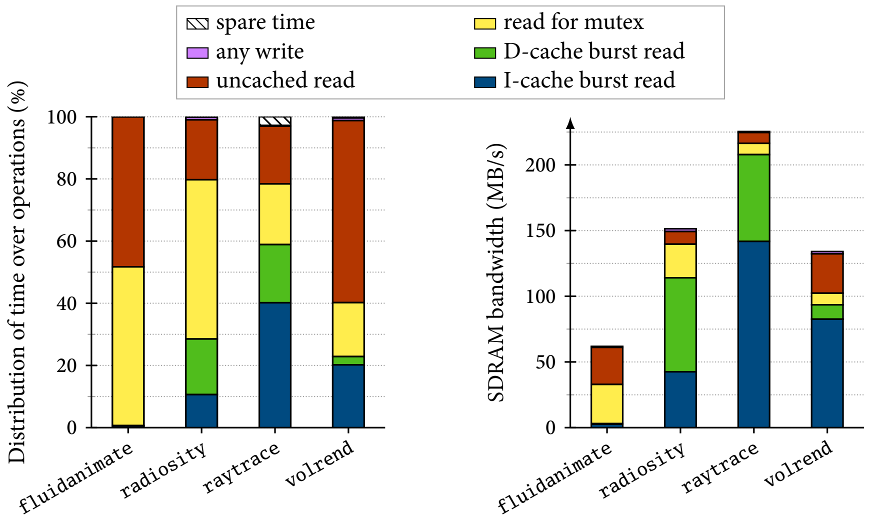
Hardware cache coherency and RMW instructions are not always applied in embedded systems, because of high hardware costs and the lack of IP [89]. Additionally, hardware cache coherency is unsupported for FPGA targets by common system-on-chip *(SoC)* design tools, such as Xilinx XPS and Altera SOPC Builder, as neither the MicroBlaze nor the Nios II supports it. The alternative is to use a generic software implementation of a synchronization algorithm, like the bakery algorithm for mutexes [69]. Again, the SDRAM is then used for synchronization, which is a scarce resource. We will bypass the memory completely by using the ring in our approach, but first investigate the effect of synchronization on the memory bandwidth using several benchmark applications.

4.4.1 Polling main memory measurements

The applications that are used in the experiments, are listed in table 4.6. The table shows the number of mutexes the applications use. `Fluidanimate` and `radiosity` have many mutexes, as a mutex only protects one (fluid) element in a grid, or a single (radiating) patch in the 3D model. The other applications use the locks in a more coarse-grained fashion, where it protects larger shared data structures. Refer to section 2.8 for more details about these applications.

Every mutex is in fact a `pthread_mutex_t`, which has been implemented using Lamport's bakery algorithm [69]. The algorithm is based on the concept of taking a unique number from a numbering machine when a customer enters a bakery shop. It works as follows. The algorithm makes use of two arrays, named *entering*, and *number*, both initialized to zero. Every process has its own element in both arrays. When a process wants to lock the mutex, it flags its presence via its entering field, and iterates over the whole number array to determine the highest number. It writes the highest number, plus one, into its field of the number array, and resets the entering flag. Then, the process polls all fields, until no other process is currently entering or has a lower number, after which it gets the lock on the mutex. It should be clear that the algorithm relies on polling the shared memory.

For the experiments, we use a 32-core variant of the system with Warpfield, as dis-



(a) by processing time

(b) by memory bandwidth

Figure 4.5 – Measured traffic on SDRAM controller

cussed in section 4.3. The memory controller of the SDRAM has hardware support to measure memory traffic. Using this information and profiling data measured by the MicroBlaze and the OS, traffic streams of the applications can be identified. Figure 4.5 plots the results. The figure presents the traffic as seen by the memory controller, which aggregates the traffic of all MicroBlazes, and it distinguishes:

» 8-word burst *instruction cache reads* (bottom of chart);

» 8-word burst *data cache reads*;

» uncached word read, participating in locking a *mutex*;

» other *uncached* word read of shared memory;

» all 8-word burst and single word *writes*;

» all *spare* time the memory controller is idle (top).

Figure 4.5(a) depicts on which operations the time is spent by the memory controller. Since the spare time is (almost) zero, the controller is completely saturated and imposes a bottleneck on the system. The bandwidth usage corresponding to figure 4.5(a) is shown in figure 4.5(b). Although the controller is saturated for all applications, the bandwidth greatly differs, because word reads and writes take almost the same amount of time as burst reads and writes, but leave most of the potential bandwidth unused. SDRAM commands like precharge and refresh do not show up in the bandwidth, but do contribute to the latency of commands.

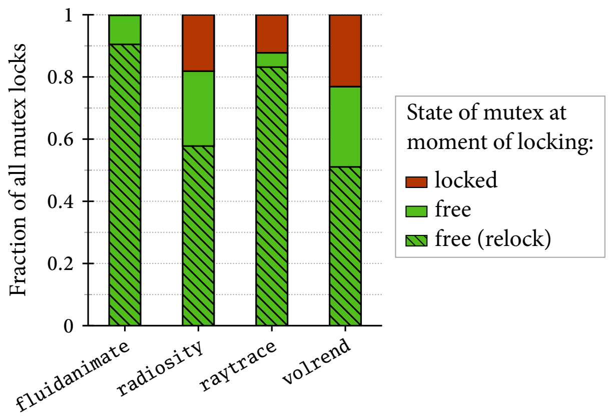


Figure 4.6 – Mutex locking behavior per application

In both figures 4.5(a) and 4.5(b), the writes are hardly visible as they occur relatively sporadically. Mutex operations contribute by 35 % on average to the memory controller load and 18 % to the bandwidth usage, surprisingly. Although the bakery algorithm relies on reads *and* writes, reads are occurring far more often than writes, because every process must poll the *entering* and *number* fields of all other processes, but writes just its own. In order to reduce the total amount of memory traffic, this mutex related traffic is a good candidate for revision, as the implementation of synchronization is transparent to the application, and can be changed without touching the application, in contrast to cache utilization and shared data accesses.

4.4.2 Mutex locality

Figure 4.6 shows additional information about the state of the mutexes of the same applications. The figure shows that mutexes of `fluidanimate` are (almost) always free at the moment they are being locked, where mutexes of the other application are free for about 80 % of the time. Additionally, the hatched area shows the fraction of mutexes that where not only free, but also classified as *relocks*: a successive lock on the same mutex by the same process.

Based on figure 4.6, it can be concluded that most of the mutexes are usually free and reused by the same process, so a mutex is (mostly) *local*. Busy mutexes, for which processes are blocking each other, are scarce. Since they involve (expensive) global synchronization, they should be avoided.

As shown above, mutexes contribute to the memory bandwidth usage, which is a scarce resource in many-core systems. The next section proposes a solution that implements locks on top of the ring, which bypasses the SDRAM completely and exploits the locality of mutexes. Since mutexes only require a small amount of memory, they can be kept locally, in a (non-coherent) SPM, for example.

4.5 Asymmetric distributed lock algorithm

As the previous section showed, synchronization traffic is a significant part of all traffic to the SDRAM. In this section, an algorithm is proposed that bypasses main memory. It is implemented on top of the core-to-core ring infrastructure, as described in section 4.3.1, and it utilizes the message-passing API of Helix (see section 2.7.2) for inter-process communication. The lock algorithm is *distributed*, because there is no single central component that handles all lock-related operations. Moreover, the algorithm is *asymmetric*, referring to the different roles of the participants in the algorithm.

4.5.1 Existing synchronization solutions

Besides using RMW operations, hardware support for synchronization can also be realized differently. Stoif et al. use a central memory controller where processors compete for protected memory regions [103]. The memory controller can also be extended to manage the state of synchronization data units that reside in the memory [82, 121]. Tumeo et al. implement synchronization using engines like the Xilinx mutex component [109]. Like fixed synchronization networks [1], all these hardware components have in common that the number of concurrent synchronization primitives is limited, where our solution scales naturally in software. Additionally, centralized units fundamentally introduce a bottleneck when scaling to more cores.

Symmetric distributed algorithms require many messages to operate [97], because all nodes need to be informed separately. These algorithms are aimed for fault tolerance, but as we do not assume a faulty device and message exchange is relatively expensive, the overhead is needlessly high. Yu and Petrov introduce a distributed lock component for every core, which all snoop synchronization messages from a bus [119]. Having such a global bus, limits scalability—the paper presents experiments with only four cores.

An asymmetric distributed mutual exclusion algorithm without experimental results has been proposed by Wu and Shu [116], using a central coordinator that forwards lock requests on mutexes, where processes form a distributed waiting queue. In this algorithm, queuing is done in a distributed manner, but locks are always sent back to the server on unlock. As we found out (see section 4.4), mutexes are often reused by the same process. This still requires many messages, where our solution optimizes for this mutex locality, and we add a quantitative evaluation.

Using local memories in a NUMA architecture for message passing is a common approach [26, 58], but generic all-to-all communication is potentially expensive in hardware. However, our write-only ring implementation can be kept low cost.

4.5.2 The algorithm: a three-party asymmetry

The ring can be kept low cost, because of three reasons. 1) The required bandwidth for synchronization purposes is low. 2) The routing in a ring is trivial, and thus cheap. 3) The latency of one clock cycle per tile is not an issue, because most of the latency is introduced by the software of the message-handling daemons—even when a hundred cores are added, the increased latency by the ring is much less than the costs of a single context switch. FPGA synthesis shows that the ring uses about 1.4 % of the total amount of logic for the PLB slaves and the ring itself. As the algorithm only relies on message exchange, which does not require time to behave correctly, predictable timing of the ring is not required.

The main idea of our algorithm is that when a process wants to enter a critical section—and tries to lock a mutex—it sends a request to a server. This server either responds with "You got the lock and you own it", or "Process p owns the lock, ask there again". In the latter case, the locking process sends a message to p, which can reply with "The lock is free, now you own it", or "It has been locked, I'll signal you when it is unlocked". When a process unlocks a lock, it will migrate it to the process that asked for it, or flag it is being free in the local administration otherwise. In this way, processes build a fair, distributed, FCFS waiting queue. In great contrast to a token-based solution, which also migrates ownership of locks, processes only give up the ownership when they are asked to do so.

Algorithms 1 to 3 show the implementation. In more detail:

- *lock server* (algorithm 1): a process that registers the owner process of a lock (or the last one waiting) using the map G. When the server gets a *request* message, it either responds with that the lock is available (line 7) or already owned (line 11), depending on whether the lock has already been registered on the server. When a process owning a lock does not need it anymore, it can *give it up*. A lock is (statically) assigned to a single server, and a server can service many locks.
- *message handler* (algorithm 2): every MicroBlaze runs a single message-handling daemon, which handles incoming messages (see section 2.7.2). When another process *asks* for an owned lock, this daemon inspects the lock administration (denoted map L_p) of the owning local process. Then, it either marks the lock for migration on unlock (line 9) or steals the lock (line 12). In case of the race condition that the give up and the ask message are sent concurrently, the daemon replies that the lock is free (line 7).
- *locking process* (algorithm 3): the process that wants to enter a critical section. When *locking*, it checks its own administration. When the lock is already owned by the process, it will lock it immediately, without communicating with other cores. Otherwise, it will request the server (line 9) and ask the owner (line 10) of the lock when appropriate. Asking for a locked lock implicitly enqueues the asking process (line 11). On *unlock*, an (uncached) read from main memory is performed (line 15), which enforces

Algorithm 1: Lock server

```
Global: administration of the owner (or last waiting in queue) of all locks
        that are currently given out by this server:
        $G$ : lock → process, $G \leftarrow \varnothing$

Procedure: request($l$,$r$)
Input: process $r$ requesting lock $l$
begin
    if $G(l)$ is unassociated then
        $G(l) \leftarrow r$
        return got lock $l$
    else
        $p \leftarrow G(l)$
        $G(l) \leftarrow r$
        return ask $p$ for state of $l$

Procedure: giveup($l$,$r$)
Input: process $r$ giving up lock $l$
begin
    if $G(l) = r$ then
        unassociate $G(l)$
```

Algorithm 2: Message handling daemon

```
Global: administration of owned locks of a local process $p \in P$:
        $L_p$ : lock → {free, locked, migrate, stolen}, $\forall p \in P_{\text{local}} : L_p \leftarrow \varnothing$

Procedure: ask($l$,$p$,$r$)
Input: lock $l$, owned by process $p$, requested by $r$
begin
    atomic
        if $L_p(l)$ is unassociated then
            return free
        else if $L_p(l)$ = locked then
            $L_p(l) \leftarrow$ migrate on unlock to $r$
            return locked
        else
            $L_p(l) \leftarrow$ stolen
            return free
```

Algorithm 3: Locking process

```
1  Global: $L_{\text{self}}$, which is one of $L$ of algorithm 2
2  Procedure: lock($l$)
3  Input: lock $l$
4  begin
5      atomic
6          $s \leftarrow L_{\text{self}}(l)$
7          $L_{\text{self}}(l) \leftarrow$ locked
8      if $s \neq$ free then
9          if request($l$,self) = ask $p$ then
10             if ask($l$,$p$,self) = locked then
11                 wait for signal (signal counterpart at line 21)
12 Procedure: unlock($l$)
13 Input: locked lock $l$
14 begin
15     dummy read SDRAM
16     atomic
17         $s \leftarrow L_{\text{self}}(l)$
18         $L_{\text{self}}(l) \leftarrow$ free
19     if $s$ = migrate to $r$ then
20         unassociate $L_{\text{self}}(l)$
21         signal $r$ (wait counterpart at line 11)
22     else if too many free locks in $L_{\text{self}}$ then
23         $l' \leftarrow$ oldest free lock in $L_{\text{self}}$
24         unassociate $L_{\text{self}}(l')$
25         giveup($l'$,self)
```

an ordering between communication via the ring and operations on the background memory, and ensures that all outstanding memory operations will be completed before the lock is unlocked—the system implements the Release Consistency memory model. This is guaranteed, as the interconnect arbitrates in FCFS manner and the memory controller processes all requests in-order. Then, only locally is the state updated (line 18), unless there is a process already waiting, which will be signaled in that case (line 21). When the process exits, all owned locks must be given up, which is left out of algorithm 3 for simplicity.

This algorithm works only when assuming that messages cannot get lost, and the message handler cannot be interrupted to handle another message. Then, mutual exclusion is guaranteed, as the first process in the queue is well known, which owns (and might lock) the mutex. When a mutex is not owned, a process requesting it only needs to send one message. If it is owned, but not locked, two messages are required. In any case, progress to enter the critical section cannot be stalled, as long as messages are handled. Moreover, the waiting time for processes that are locking a mutex, is bounded by means of this queue—assuming that the process that holds the lock, will release it eventually.

Unfortunately, testing the algorithm with a model checker like Spin [56] is not feasible. To test queuing properly, the Spin model requires many concurrent processes, which all have multiple ways of interleaving messages and state changes. This leads to excessive verification run times. On the other hand, when the model is simplified in order to reduce verification time, it is unclear whether it still matches the algorithm.

Next, the performance of the proposed ring and new lock algorithm will be compared to the bakery lock.

4.5.3 Experimental comparison results

To evaluate the distributed lock, experiments are conducted on the same 32-core system as of section 4.4. The bakery lock is compared to the distributed lock, where the former is referred to as the 'base case'.

For the latter, the maps G and L_p of algorithms 1 to 3 are implemented using AA-trees [8], having $\mathcal{O}(\log |M|)$ complexity, where M is either map. The maps are not bound in maximum size. Every node in the AA-tree, i.e. mutex data structure, of G consumes six words of heap memory, every node of L_p uses nine words. Although the different concepts of a lock server and message-handling daemon are important in the algorithm, the functionality of the server is merged into the daemon in the actual implementation. This allows a quick response of request and give-up messages. As a result, every message-handling daemon can act like a lock server, and locks are statically assigned to one of the 32 daemons based on the address of the mutex, which implements a naive way of load balancing.

The four applications have been run for both configurations, repeated ten times with slightly different compile settings to average out cache effects by placing memory segments differently. All applications start one worker process on each of the 32 cores. When a process blocks on a mutex, the blocked time is left unused by the application for that specific core. Only the parallel body of the application has been measured; (sequential) preprocessing steps have been ignored.

Figure 4.7 shows the types of memory traffic of both the base case and the distributed lock. The figure shows that for `raytrace` and `volrend` the memory bandwidth is not saturated. In table 4.7, the measured number of exchanged messages is depicted. Even for `fluidanimate`, the ring utilization is very low, although the application is quite message-intensive—every core handles 809 messages per second on average. One message consists of six words, where the ring allows injection

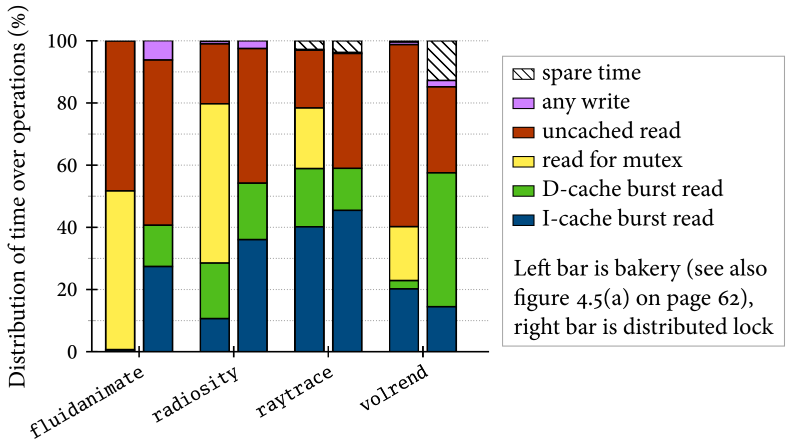


FIGURE 4.7 – Measured traffic on SDRAM controller, using bakery and distributed locks

TABLE 4.7 – Distributed lock statistics

	# locks	request[a]	ask[a]	giveup[a]	signals[a]	msgs/s[b]
`fluidanimate`	2 935 247	285 176	280 665	4 511	1 110	25 898.3
`radiosity`	1 594 306	627 154	574 620	52 533	214 764	8 480.6
`raytrace`	33 831	1 681	1 645	36	73	102.0
`volrend`	56 380	33 962	33 886	76	8 024	3 892.9

[a] See algorithms 1 to 3
[b] Lock messages exchanged per second over the ring

of one word per core every clock cycle. Thus, the ring has at least a bandwidth of $\frac{1}{6} \cdot 100 \cdot 10^6$ words/s = $16.7 \cdot 10^6$ messages/s, of which `fluidanimate` uses 0.155 %.

Performance numbers, which are averaged over all runs, of the applications are shown in figure 4.8 on the next page. All values are normalized to the base case with the bakery algorithm, which is 1 by definition. In the chart, the first metric shows the relative change of the execution time of the application: all application benefit from the distributed lock, `fluidanimate` is even 9 times faster using the distributed lock compared to using the bakery lock.

Next, two metrics indicate the SDRAM usage, which aggregates operations of all 32 cores. It shows that the memory bandwidth is used more effectively; the read bandwidth increases, with a similar time spent on reading, because less uncached word and more burst operations are performed.

Finally, two metrics are shown of one MicroBlaze that has hardware support for measuring microarchitectural events. Since all cores are running the same kind of workload, it can be assumed that all other cores behave similarly. Overall, the core

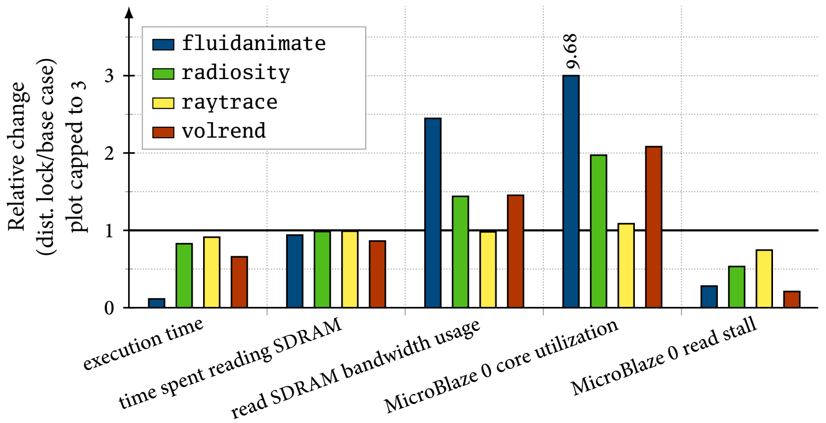


Figure 4.8 – Difference between using bakery algorithm and distributed lock with ring

utilization is higher for all applications, since cores stall less on (uncached) reads.

4.5.4 Locality trade-off

Whether the complexity of the distribution lock pays off for a given application, is closely related to the locality of its mutexes. There is a trade-off between a main-memory polling algorithm like bakery that consumes scarce bandwidth, or keeping (or caching) mutexes locally and having higher latencies for non-local mutexes. Figure 4.9 gives insight into this trade-off.

Naturally, when a mutex is used by only one process, it is always local and locking it is very fast. When mutexes are used by more processes, the lock must be migrated regularly, which involves communication between cores. Hence, the amount of expensive communication depends on the *locality* of the mutex, which we define as the fraction of relocks over free locks. In the synthetic setup used for figure 4.9, a mutex is forced to a specific locality, and the average time is measured of locking that mutex. The figure shows the relation between locality and average lock time for both the bakery implementation (which takes 270 µs on average) and the distributed lock (3.5 µs for a relock). Although the exact slope and height of the lines in the figure depend on the workload, the trend is always the same.

For the four applications, the locality (as can also be found in figure 4.6 on page 63) is respectively 0.91, 0.71, 0.95, 0.66, and is also indicated in figure 4.9. This shows that applications with globally used mutexes, might not benefit from the distributed lock[3], but the tested applications are all at the right side of the break-even point.

[3]Obviously, one can argue that having global locks in massively parallel applications is a bad programming habit anyway.

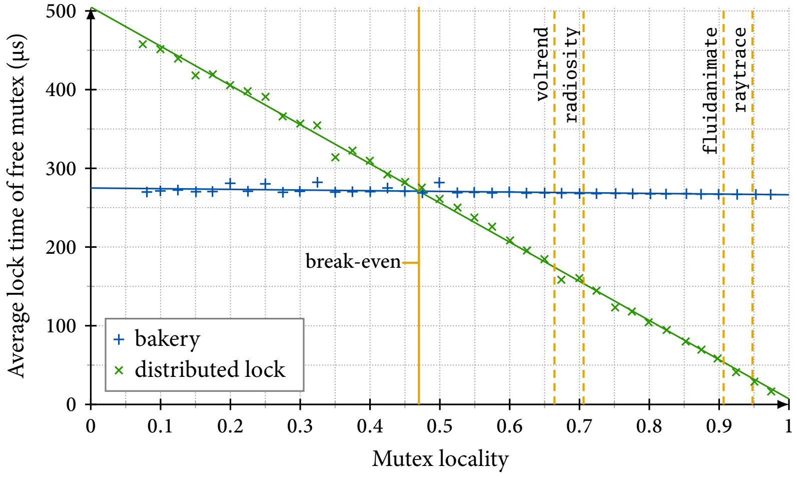


Figure 4.9 – Impact of locality on acquiring time of a lock, based on synthetic benchmark

Table 4.8 – Distributed lock performance measurements, as fraction of total execution time

	lock (%)	waiting (%)
`fluidanimate`	7.58	27.42
`radiosity`	1.28	24.72
`raytrace`	0.03	0.19
`volrend`	0.50	8.74

Table 4.8 shows two additional measurements regarding locking behavior of the distributed lock. It lists the time every application spends on executing the locking algorithm (including sending and waiting for messages) while trying to get a lock on a mutex, and on waiting for a locked mutex. For example, `fluidanimate` spends in total 35 % of the whole execution time of the application on locking operations. Since most of the locking time is spent on waiting, improving the locking algorithm itself any further will hardly lead to a performance increase; making locks more local is probably more beneficial.

Although the distributed lock has only been tested on 32 cores, we expect that the costs and trade-off are the same on larger systems. As algorithms 1 to 3 do not depend on the number of cores, just the number of stored locks in maps G and L_p influence the performance. When the balance between servers and worker processes is kept the same, then the concurrency in the design of the application is the only relevant factor.

4.6 Hardware and performance scalability

This chapter proposed two optimizations on existing systems: a new interconnect and a new distributed lock algorithm. Although these solutions are designed to be scalable, this section will discuss in more depth how the application's performance is influenced when scaling to more cores. We evaluate the total performance in terms of the combined amount of executed instructions, with respect to the number of cores, denoted n. Ideally, doubling the number of cores will also double the total computing power, and therefore the performance. Obviously, this is in practice not the case. In general, adding a core will increase traffic to the main memory, which has a limited bandwidth, resulting in slowing down other cores.

The arbitration tree of the Warpfield interconnect handles the core-to-memory traffic. As section 4.3.1 discussed, the required amount of hardware resources is constant per core, for both the core and tile, as for the tree. Therefore, the hardware scales $\Theta(n)$.

However, the performance is more complex to determine. The tree grows in number of multiplexers, so the depth of the tree will grow $\Theta(\log n)$ (see also equation 4.1). Therefore, memory reads will take more time to complete, when the number of cores is increased. Additionally, more cores also means more concurrent traffic streams to memory. For simplicity, we assume that the MicroBlaze executes every clock cycle one instruction, which is to be read from memory, and every eighth instruction actually reads data from memory. So, executing eight instructions reads the memory nine times, but most of the reads are either an instruction or data cache hit. We denote the weighted average of the instruction and data cache hit rate as α, and we assume that cache lines are filled using the target-word-first policy, such that the MicroBlaze can continue when the first word from memory arrives.

Assume that the system is not bounded by the memory bandwidth, so concurrent reads of different MicroBlazes do not obstruct each other. Then, the sum of executed instructions of all cores per second can roughly be calculated as

$$\frac{\text{number of instructions, such that one read miss occurs}}{\text{latency of one miss + one cycle per hit}} \cdot 100\,\text{MHz} \cdot n.$$

Given a cache hit rate of α, a cache miss occurs $1-\alpha$ times per memory read. In other words, $\frac{1}{1-\alpha}$ reads will generate one cache miss and $\frac{1}{1-\alpha}-1$ hits. As discussed above, the instructions–reads ratio is $8:9$. Moreover, an instruction and data cache hit might occur simultaneously. Using equation 4.1, and an SDRAM memory controller latency of 15 cycles, we can fill in the equation:

$$\simeq \frac{\frac{8}{9}\cdot\frac{1}{1-\alpha}}{(S_{\text{bc}}(\text{read}) + \text{SDRAM latency} + \text{back via tree}) + \left(\frac{8}{9}\cdot\left(\frac{1}{1-\alpha}-1\right)\right)} \cdot 100\,\text{MHz} \cdot n$$

$$\simeq \frac{\frac{8}{9}\cdot\frac{1}{1-\alpha}}{(1+\lceil\log_2 n\rceil) + 15 + \lceil\log_2 n\rceil + \left(\frac{8}{9}\cdot\left(\frac{1}{1-\alpha}-1\right)\right)} \cdot 100\,\text{MHz} \cdot n$$

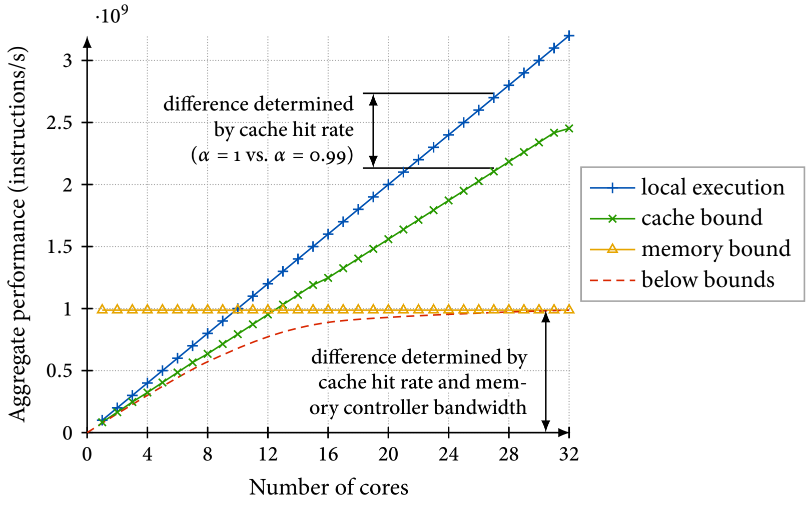


FIGURE 4.10 – Scaling trends of Starburst, at 100 MHz, cache hit rate of $\alpha = 0.99$, and memory bandwidth of $1.11 \cdot 10^7$ reads/s

$$= \frac{\frac{8}{9} \cdot \frac{1}{1-\alpha}}{2\lceil \log_2 n \rceil + \left(\frac{8}{9} \cdot \left(\frac{1}{1-\alpha} - 1\right)\right) + 16} \cdot 100\,\text{MHz} \cdot n. \tag{4.5}$$

Its complexity is bounded by

$$\begin{aligned} &\Theta\left(\frac{\frac{1}{1-\alpha}}{\log n + \frac{1}{1-\alpha}} \cdot n\right) \\ =\ &\Theta\left(\frac{n}{(1-\alpha)\log n + 1}\right). \end{aligned} \tag{4.6}$$

Equation 4.6 depends on the number of cores and the cache hit rate. It shows that when the every read is a hit, so $\alpha = 1$, the number of executed instructions scales linearly to the number of cores. Figure 4.10 visualizes these trends. The figure shows both the linear speedup, when no cache misses occur and all memory accesses are local, and the trend when we assume that the performance is only bound by the cache hit rate, with infinite memory bandwidth. Although the trend that is bounded by the cache hit rate, seems to be linear, equation 4.6 shows that it is not. The relative distance between the two trends is increasing with a larger number of cores, although the impact on the performance only becomes dominant after millions of cores when having a high cache hit rate.

In practice, the memory bandwidth is limited. When the number of cores is increased, there is more memory traffic, and the memory controller will saturate at some point. Then, the bandwidth of the memory controller determines the maximum number of reads, which is related to the maximum number of executed instructions, given a specific cache hit rate and an instructions–reads ratio. So, given a memory bandwidth in terms of number of memory reads, the number of executed instructions of all cores combined can be calculated as

$$\begin{aligned} &\text{maximum number of memory reads} \cdot \text{instructions per cache miss} \\ \simeq\ &\text{memory bandwidth} \cdot \left(\frac{8}{9} \cdot \frac{1}{1-\alpha}\right). \end{aligned}$$

The bound by the memory bandwidth depends on both the available memory bandwidth and the cache hit rate. The performance does not depend on the number of cores. So, when the application is memory bounded, increasing the number of cores does not improve the performance; all cores will be slowed down, which exactly counteracts the increase in raw computing power. This bound is also visualized by figure 4.10. Obviously, the actual performance of the application is bounded because of both the cache hit rate and the memory bandwidth, of which an indicative dashed line is given in the figure.

Synchronization, as realized by the distributed lock algorithm, does not use the arbitration tree. It uses message passing over the ring instead. The ring has a latency of one clock cycle per tile, so it scales $\mathcal{O}(n)$. Let $|L|$ denote the total amount of locks in the application, then every core serves $\frac{|L|}{n}$ locks when the locks are properly balanced over the servers. Moreover, every lock has a constant administration overhead, so the required memory is bounded by $\Theta\left(\frac{|L|}{n}\right)$ per server. To process one lock-related message, the latency is bounded by

$$\begin{aligned} &\mathcal{O}\left(\text{ring latency} + \text{AA-tree lock administration latency}\right) \\ =\ &\mathcal{O}\left(n + \log\frac{|L|}{n}\right). \end{aligned} \tag{4.7}$$

In practice, the latency of the ring is negligible, as a context switch alone to start the message handler requires hundreds of clock cycles, and the uncertainty of a few cache misses mitigates the influence of the ring latency. The number of locks, how they are distributed, and their locality depend largely on the application. Hence, whether the application scales properly and how it uses the parallelism of the platform to distribute or balance work, cannot be determined in general.

4.7 Conclusion

This chapter presented two improvements regarding the interconnect and synchronization. Experiments show that by replacing the connection-oriented Æthereal by the connectionless Warpfield NoC, the execution time of a set of SPLASH-2 and PARSEC benchmark applications is reduced by 27 % on average, even though the analytically determined worst-case latency bound of the NoC increased. Using the asymmetric distributed lock instead of a polling-based bakery lock completely eliminated memory traffic for locks, by utilizing a low-cost core-to-core interconnect.

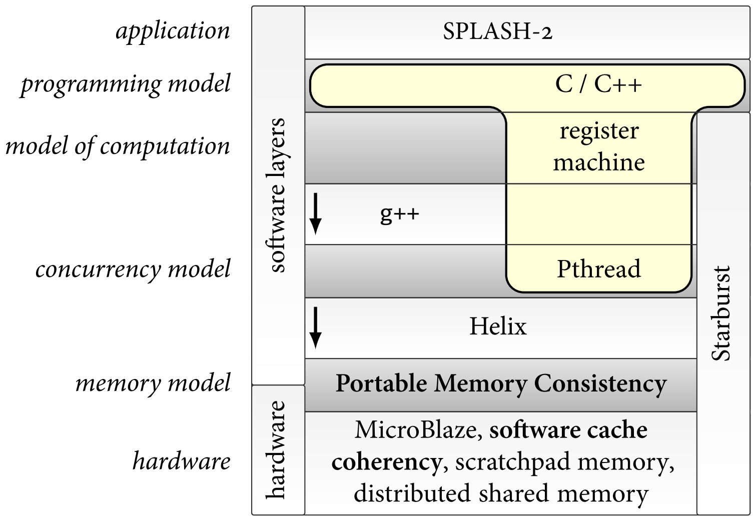
This gives an additional performance boost of 37 % on average, which exploits the locality of mutexes.

Warpfield's hardware costs scale linearly to the number of cores, and the application's performance scales close to linear, assuming a high cache hit rate. However, the performance will inevitably be bounded by the memory bottleneck, after which the performance does not change anymore when even more cores are added.

With respect to the overview figure on page 44, these results are the effect of modifications to the platform, which are transparent to the applications. The value of these modifications lies in the assumptions made by the programming model: threaded C programs are memory-oriented and hardly constrained in memory access patterns. Æthereal is not tailored to such a specific communication pattern, but focused towards predictability, given an application with KPN-like concurrency.

As discussed earlier, threading and C is a common approach to programming multiprocessor systems. Threading implies using mutexes, which has an associated cost, so high contention on a mutex slows down the application. To prevent contention, applications should be designed such that sharing a mutex is kept to a minimum, which results in a high locality of mutexes. As the benchmark applications have such high locality, they benefit from the distributed lock. As contrasting example, a KPN-based program might not use mutexes at all, in favor of FIFO channels. This shows that the programming model puts a specific stress on aspects of the platform—applying the distributed lock algorithm in a platform that only uses KPN-based programs, does not improve the performance.

In the case of the distributed lock algorithm, the platform defines that the performance of an application increases when locks have a high locality. Hence, the platform associates costs to low-level operations. However, only when the programming model can make use of the cheap, i.e. efficient, low-level operations, one can speak of an efficient system, otherwise this hardware efficiency is useless. In other words, it is the programming model that gives a *meaning* to the efficiency of hardware.



Chapter 5 overview

Usable Weak Memory Model

Abstract – *Porting software to different platforms often requires modifications of the application, when the supported programming model is different. Commonly, different platforms support different memory consistency models. In this chapter, an approach is presented that makes applications independent of the memory model of the hardware. As a result, they can be compiled to hardware that supports any of the common memory architectures. The key is having a synchronized weak memory model that only guarantees the most fundamental orderings of reads and writes, and annotations to specify additional ordering constraints explicitly. As a result, tooling can transparently and properly implement fences, cache flushes, etc. when appropriate, without losing flexibility of the hardware design.*

With the growth in the number of mobile and embedded devices, porting software to various platforms is becoming increasingly important. Programmers not only face different software contexts (OSs and APIs), but also different hardware architectures with various numbers of cores and communication infrastructures. Porting[1] to other hardware often requires subtle, but fundamental changes to the software, due to a changed memory consistency model. As section 2.6 discussed, commercial many-core systems assume being programmed using C, and C includes the memory model in the programming model. Therefore, the application has to be adapted to changes in the memory model and thus the programming model, which can be a thorough and error-prone task.

In section 3.6, we concluded that porting an application could only be done transparently, when the programming model does not change. This chapter will remove

Large parts of this chapter have been published in [JHR:6].

[1] *Porting* means translating a program such that it can be run on another platform. *Portability* is a property, which means that porting such a program is easy.

the memory model from the threaded C programming model, which is visualized in the overview figure on page 76. Application will have to be modified once to support the memory-model-less programming model, but the application will become portable to any hardware, regardless of the actual memory model of the hardware. This approach is in contrast to the optimizations discussed in chapter 4, which were applied to the platform transparently.

To this extent, this chapter presents Portable Memory Consistency *(PMC)*, which defines a memory model (referred to as the *PMC model*), and an approach to apply this model to an application and any memory architecture, by means of annotations to the source code (the *PMC approach*). Traditionally, a memory model is seen as a contract between hardware and software, and defines the semantics of reads and writes. In contrast, we use our memory model as an *abstraction layer* that disconnects the application from the underlying hardware. The key is that all orderings that are required by the application, are made explicit—the abstraction contains more details than its implementation. Then, (glue) tooling can fill in the gap between what the application requires and which orderings are already satisfied by the hardware. As a result, porting applications to hardware with another memory model becomes just a compiler setting.

For this, we propose a single, weak, synchronized memory (consistency) model that only defines five memory operations and four types of orderings between them. This model 1) is strong enough to mimic Sequential Consistency when required by the application; 2) is weaker than Entry Consistency, because synchronization operations to different memory locations are unordered, unless explicitly specified by fences; and 3) allows mapping to all existing hardware, because it is an intersection of all common memory models. (We will discuss these models in section 5.1.1.) Since changing a memory model of an existing programming language is impossible—we use C and C++ in our experiments—it is required that the source code is annotated to indicate which orderings are required by the application[2].

The PMC approach involves that an application is designed and annotated for the PMC model, regardless of the targeted hardware. The PMC model is designed such that a mapping of the primitives and ordering relations to specific hardware can be designed and verified with relative ease. Since all required orderings are made explicit, the platform can use this information to take all measures in either software or hardware to ensure the orderings and synchronization on the hardware at hand, without losing flexibility of optimization of other non-ordered operations. The approach is evaluated based on case studies with three memory architectures.

5.1 The problem with memories

Porting software to hardware with another memory model can cause very subtle problems. Listing 5.1 shows an example of this. The program of the figure intends

[2] Although using the PMC model natively in the semantics of a new programming language is the best way to go, this is left as future work.

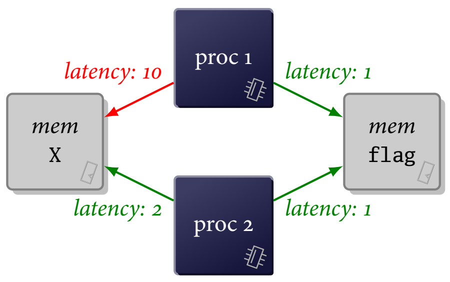
Initially: `flag=0`

Process 1:

```
X = 42;
flag = 1;
```

Process 2:

```
while(flag!=1)
    sleep();
print(X);
```



Listing 5.1 – A Sequentially Consistent correct program, which breaks on an architecture with two memories

to communicate the value 42 from process 1 to 2 via variable `X`. On a platform that implements Sequential Consistency, this program will behave correctly.

However, the program will break when it is run on a hardware architecture that is also depicted in listing 5.1. The essence of the problem is that the latency of the write operation by process 1 to the memory that holds `X`, is higher than that of `flag`. When process 2 polls the `flag`, it first reads `flag` being 1 and then reads `X`. Because of the high latency of the write of `X`, process 2 can read the old value of `X` before 42 has arrived in the memory—the program breaks. Tracking down this bug is non-trivial by looking at the source code, and could even be more difficult to find when the latencies in the interconnect vary over time. The problem cannot be prevented, even if both `X` and `flag` are declared `volatile`, atomic or separated by fence instructions.

The underlying problem in this architecture is that the order of the two writes of process 1 is not guaranteed, as is the case for Sequential Consistency. The behavior of the memory—which is distributed in this example—is defined by a memory (consistency) model, which prescribes the conclusions a *process* can draw when it *observes* state changes of *locations* of the memory and whether different processes must *agree* on these conclusions (see also section 3.2). In the example, the conclusion that every process agrees that 42 is visible before the `flag` is set, is wrong, even though the write of `X` is initiated first. Numerous memory models have been proposed throughout the years, of which we will discuss several next.

5.1.1 Various memory models

Memory models can be grouped in two classes: uniform and synchronized. Uniform models have only two operations on the memory, read and write, whereas synchronized models define additional special operations options, usually acquire and release. Figure 5.1 on the next page presents a taxonomy of several models. What all models have in common is that all control and data dependencies local to a process are preserved; a process will always see changes to its variables as the program prescribes. The differences of the models lie in how processes see each

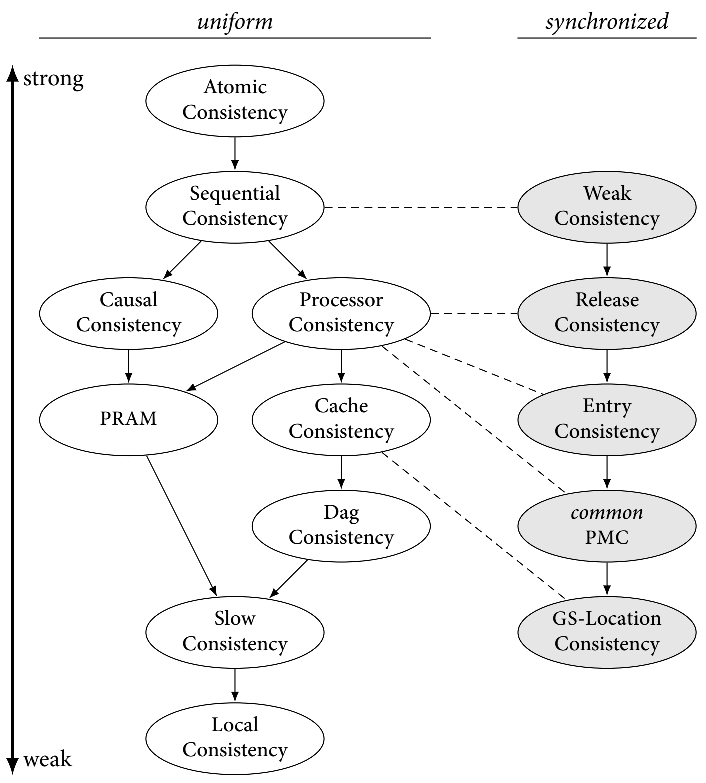


FIGURE 5.1 – Taxonomy of several memory models. Arrows indicate strictness ordering, dashed lines indicate the equivalent strictness of synchronization operations.

other's writes. We will discuss several models in an informal way to get a grasp about the range of differences among them.

As discussed before, Sequential Consistency *(SC)* [70] defines that all operations are in a single (possibly run-time dependent) total order. That means that every process(or) will agree on the order in which all state transitions occurred. Refer to listing 5.2 for an example of a possible interleaving. The figure shows three processes, executing read and write operations on two shared variables X and Y. The processes do not have a control flow that determines the order in which operations of different processes are interleaved. A trace in the figure shows per process when operations are executed in time, which progresses from left to right. So, every trace shows a specific interleaving of writes, e.g., Y=2, and reads, e.g., a==X==1, which means that X is read and happens to be 1, and is stored in local variable a. Trace 5.2(a) is a valid execution under the SC model; all processes agree that X and Y are written in the following sequence: Y=2, X=1, Y=1, and X=2.

In Processor Consistency *(PC)* [6, 83], processes must agree on the writes of one

Initially:
all set to 0

Process 1:

```
X=1;
a=X;
b=Y;

Y=1;
```

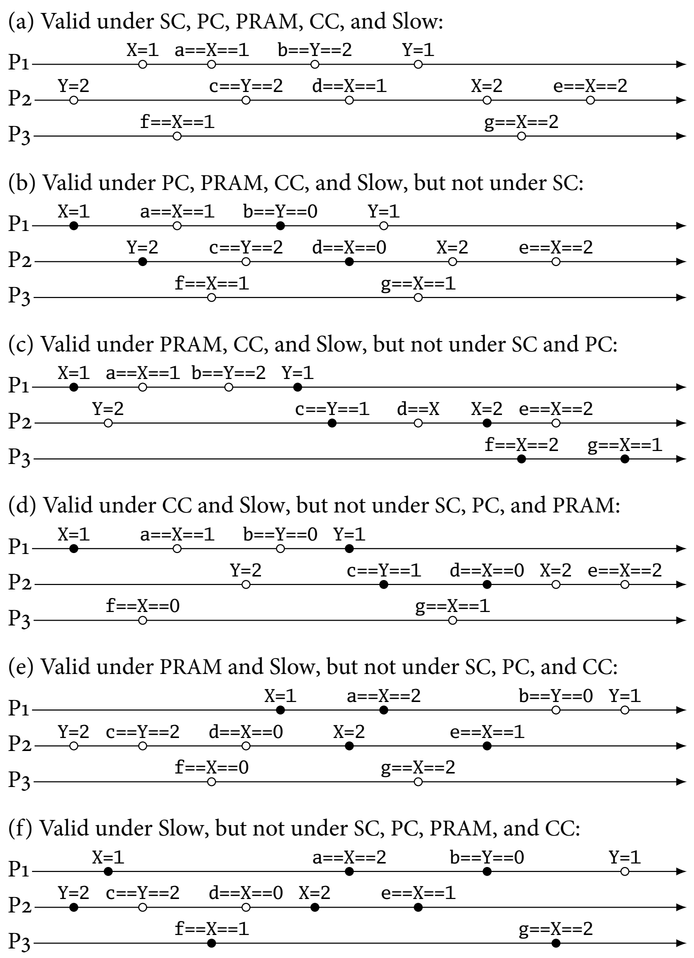
Process 2:

```
Y=2;
c=Y;
d=X;

X=2;
e=X;
```

Process 3:

```
f=X;
g=X;
```


(a) Valid under SC, PC, PRAM, CC, and Slow:
P1 X=1 a==X==1 b==Y==2 Y=1
P2 Y=2 c==Y==2 d==X==1 X=2 e==X==2
P3 f==X==1 g==X==2
(b) Valid under PC, PRAM, CC, and Slow, but not under SC:
P1 X=1 a==X==1 b==Y==0 Y=1
P2 Y=2 c==Y==2 d==X==0 X=2 e==X==2
P3 f==X==1 g==X==1
(c) Valid under PRAM, CC, and Slow, but not under SC and PC:
P1 X=1 a==X==1 b==Y==2 Y=1
P2 Y=2 c==Y==1 d==X X=2 e==X==2
P3 f==X==2 g==X==1
(d) Valid under CC and Slow, but not under SC, PC, and PRAM:
P1 X=1 a==X==1 b==Y==0 Y=1
P2 Y=2 c==Y==1 d==X==0 X=2 e==X==2
P3 f==X==0 g==X==1
(e) Valid under PRAM and Slow, but not under SC, PC, and CC:
P1 X=1 a==X==2 b==Y==0 Y=1
P2 Y=2 c==Y==2 d==X==0 X=2 e==X==1
P3 f==X==0 g==X==2
(f) Valid under Slow, but not under SC, PC, PRAM, and CC:
P1 X=1 a==X==2 b==Y==0 Y=1
P2 Y=2 c==Y==2 d==X==0 X=2 e==X==1
P3 f==X==1 g==X==2

Listing 5.2 – Interleavings of operations under different uniform memory models. Particularly interesting sequences of operations are marked by black dots.

process, and on all writes of all processes to the same variable. However, processes can disagree on the order of writes to different locations by different processes. Consider trace 5.2(b). Several operations are marked as black dots. Focus on these black operations in the trace, as the other operations are not relevant for the example. Process 1 observes that X has been written before Y, as it reads the initial value of Y after it wrote X. Process 2 observes exactly the opposite. Therefore, the processes disagree on the interleavings of the two writes to X and Y. Under SC, this would not be valid, but as PC does not define an order of the two writes of the different processes, it is a valid outcome. Hence, PC is weaker than SC. This trace only shows one case where PC and SC differ, but many more examples could be constructed.

The even weaker Cache Consistency *(CC)* [101] defines that all writes to the same variable should be observed in a total order, regardless which process wrote it. Take a look at trace 5.2(d), and focus again on the black operations. In this trace, X's values will follow the sequence 0, 1, 2. Process 1's writes, X=1 and Y=1, are observed by process 2 in a different order than process 1 wrote it; it first reads Y being 1, and then sees X being 0, so process 2 can conclude that Y was written first. This is allowed under CC, because CC does not define a relation between operations on different locations. Since this trace is not valid under PC, CC is weaker than PC.

In contrast to CC, Pipelined RAM *(PRAM)* [74] defines that all writes of the same process should be agreed on instead. The interleaving of writes by different processes is not defined. Trace (e) gives an example of process 1 and 2 disagreeing on the last write to X—process 1 sees X become 2 after it wrote X, where process 2 sees the opposite. Therefore, there is no single order in which the writes happened, so 'the' value of X is undefined. CC and PRAM are both weaker than PC, but their mutual relation is not defined. This is also depicted in figure 5.1 on page 80.

One might conclude that PC is the combination of CC and PRAM. However, in its original definition, trace (c) gives a counterexample. The total order on X, as observed by process 3, turns out to be the sequence 0, 2, and then 1. However, there is a causal relation via Y and process 2 that suggest that this sequence cannot exist, but CC and PRAM do not prohibit this. (Note that the actual value in d might render this trace invalid for either CC or PRAM, depending on whether 0 or 1 is read. Therefore, the value is left out of the trace on purpose.) Whether this peculiar situation was intended originally, is debatable [101].

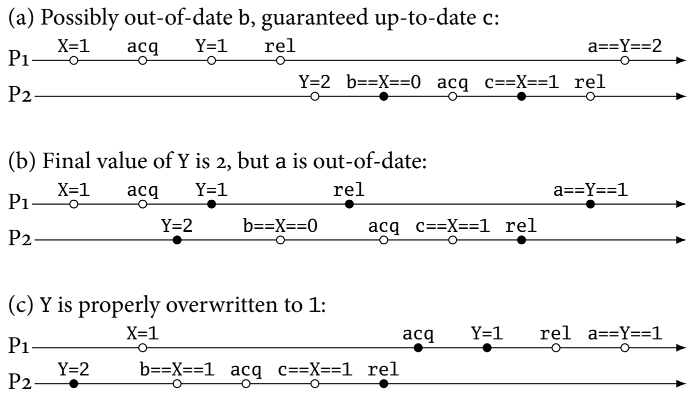
Finally, the weakest model in our discussion is Slow Consistency [59], where only the order of operations of one process to the same variable is guaranteed. So, there is a notion about older and newer values, but processes can disagree on the relation between writes of two processes and writes to different variables. Imagine this model as that values are distributed 'slowly' through the system, and every process receives all writes eventually, although updates are delivered depending on the distance of the writer and the memory location. Therefore, when a 'newer' value is read, a successive read will not return an 'older' value anymore. Trace (f) exemplifies a valid, but hard to use, interleaving of operations: process 1 and 3 disagree on the order of writes to X, and process 1 sees the writes to Y and X of process 2 differently.

Synchronized models are more complicated than uniform ones, because there is usually a difference in the guarantees about ordering of ordinary reads and writes, and the special synchronization operations. Steinke and Nutt [101] formalized synchronized models as that they have transitions between two different uniform models: a model for reads and writes, and a (usually stronger) model for the synchronization operations. Figure 5.1 illustrates this relation. The synchronized models have in common that reads and writes are behaving under Slow Consistency. A relatively strong synchronized model is Weak Consistency [41], which has only one synchronization operation. This operation is ordered like SC, and it forms a barrier for reads and writes before and after it.

```
Initially:
all set to 0

Process 1:
X=1;
acquire;
Y=1;
release;
a=Y;

Process 2:
Y=2;
b=X;
acquire;
c=X;
release;
```



Listing 5.3 – Valid interleavings of operations under Release Consistency

The acquire and release of Release Consistency *(RC)* [45] behave in a similar manner as reads and writes, respectively, of one variable under PC. The writes of one process only have to be visible for others after it has executed a release, and writes of other processes only have to be observed after an acquire. As a result, updates to the memory only have to be communicated upon acquire from the process that did the last release. Listing 5.3 gives several possible traces.

Where RC's acquires and releases are unrelated to specific shared variables, Entry Consistency *(EC)* [12] does make this differentiation. Listing 5.4 on the next page gives an example code that has some similarities to listing 5.3. Like PC defines a per-process and per-variable ordering, EC does the same with acquires and releases per variable; acquires and releases executed by one process are observed in the order they are executed, regardless on which variable they operate, where acquires and releases on one variable form a total order over all processes. Moreover, EC requires that all accesses to variables are wrapped either by acquire–release pairs with exclusive access to the variable, or (read-only) non-exclusive pairs. The non-exclusive pairs may overlap with other non-exclusive pairs, whereas exclusive access cannot overlap with any other pair operating on the same variable. Naturally, the state of the variable that is observed within a non-exclusive pair, is always the state as of the last exclusive access; reads can never be out-of-date, as it can be the case with RC.

A synchronized weak memory model that is tailored towards the streaming application domain, is Streaming Consistency [110]. This model assumes that processes communicate via (circular) buffers. An acquire–release pair is related to a buffer, and protects accesses to it, regardless whether it is read or written. Corresponding to EC, acquires and releases are ordered like PC, and accesses to different buffers by different processes are not ordered. Because the model makes writers and readers

Process 1:

```
acquire(X); // exclusive access
X=1;
release(X);
acquire(Y); // exclusive access
Y=1;
release(Y);
acquire_ro(Y); // non-exclusive
a=Y;
release_ro(Y);
```

Process 2:

```
acquire(Y); // might overlap
Y=2;        //  with acquire(X)
release(Y);
//b=X; // not allowed outside of
       //  acquire-release pair
acquire_ro(X);
c=X;
release_ro(X);
```

Listing 5.4 – Example source code for Entry Consistency

of a buffer explicit, it allows reasoning about functional behavior more easily.

Other weaker models exist, but their usability is limited. For example, GS-Location Consistency [44] is one of the weakest synchronized models, but Long et al. [75] point out that specific algorithms cannot be implemented. Although their proposed solution is formally correct, it is impractical to implement because of global dependencies, and it breaks the cache coherency protocol of the original paper.

The PMC model can be characterized as a synchronized memory model, which is weaker than EC, but still has its acquire and release operations ordered corresponding to reads and writes of PC. As the reader might have noticed in understanding the traces of listing 5.2, reasoning about memory models is notoriously hard. Therefore, the memory model should be as strict as possible to make reasoning about it easier, but not stricter than required by the application to allow maximum freedom of optimization in the platform. We believe that PMC is suitable in this context.

5.1.2 PMC's basic idea

The basic idea of our approach is that there are as few implicit constraints of ordering of operations as possible, and that all additional constraints should be defined explicitly in the source code. So, the solution is twofold: a weak memory model, and annotations for additional constraints. This memory model, which will be discussed in more detail in section 5.2, can be summarized as that it is only guaranteed that reads and writes from the same process(or) to the same location will be observed in the same order. Additionally, the annotations allow a compiler to insert special memory operations, which enforce an order between two operations of different locations by one process (a proper fence), and two operations on the same location by multiple processes (acquire/release).

Regarding the example of listing 5.1 on page 79, if the source code indicates that the write to `X` and `flag` should be observed in that specific order, then a compiler or OS can enforce it. For example, a compiler can insert a read of `X` between the writes to `X` and `flag`. Since the read completes after `X` has been written, it is guaranteed that every other process will first observe the change to `X` and then `flag`.

PMC essentially splits the memory model in two layers: one memory model abstraction, which is shared among all applications, and the actual memory model of the hardware. As a consequence, the strictness of the hardware's memory model becomes just a feature. This is similar to having hardware floating-point support in a processor: a programmer can always use floating-point operations in an application, but computation is faster when the hardware supports it (at the cost of increased chip area), otherwise software emulation is used. Similar, synchronization can always be used, e.g., by using a bakery lock, but when the memory model of the hardware is stricter, atomic RMW operations are faster (at the cost of overly constraining other possible operation interleavings).

In literature, memory models and their usability have been studied widely. As discussed in chapter 3, the main motivation for defining different weak memory models is to achieve efficiency of the hardware implementation. Nevertheless, these models have a strong mathematical basis. Most work focuses on the memory model itself and, to the best of our knowledge, no work directly relates such a formalism to how it is implemented in hardware and used by applications in practice. For example, memory models require that the source code is properly labeled (in other words, annotated) [45], but do not discuss in detail how the annotation should be used. In contrast, PMC links the memory models to annotations in the source code and to the implementation on concrete hardware.

Steinke and Nutt [101] analyze memory models, and give a taxonomy that is based on the models' common properties. They discuss thirteen uniform models (and conclude that there can be more). Their discussion focuses on formal properties, which do not (easily) allow an implementation. In contrast, we describe a concrete implementation of the memory model we present in this chapter.

Integration of a memory model in a programming language is preferable, such that tooling can verify or complement ordering constraints. The latest C++ standard (C++11 [24]) includes multithreading and defines a memory model. It assumes that the programmer can identify variables that should be declared atomic and access it accordingly. However, Batty et al. [11] conclude that this model is not clearly defined by the standard, and the corresponding mathematical model might not be 'sufficiently widely accessible'. Because of the complexity of the model, it is unclear whether it defines the weakest (usable) model. Therefore, it is also unclear whether maximum freedom in the execution is allowed, and whether maximum performance can be achieved because of that. We define a memory model that we will argue to be the weakest usable model, which simplifies reasoning about behavior, and is practical to implement.

Hill [55] argues that multiprocessor systems should implement sequential consistency, because the performance increase by relaxed models does not justify the added complexity for 'middleware authors'. However, the paper does not address many-core systems and scaling issues.

We discuss the memory model, annotations and implementations in more detail in the next three sections.

5.2 A Portable Memory Consistency model

In this section, we present a synchronized weak memory model. This model is the programmer's view on memory in the PMC approach.

5.2.1 Fundamentals

A program defines a partial order of operations, such as reads and writes, on memory locations. This order of operations can be represented as a directed acyclic dependency graph. This section will define the properties of such a graph. In general, different concurrent processes can observe operations in a different order. However, the edges in the graph indicate which operations are ordered in time, independent of who observes them. These dependencies can partly be determined at compile time, but some parts are only known at run time, due to data dependencies and control flow, for example. At run time, all dependencies are known—although such a graph is never actually stored. One can see this graph as the complete history of the state of the memory. Such a state at run time is an *execution* of a program. For the base model, we use a notation that is similar to the one as proposed by Steinke and Nutt [101].

Definition 5.1 (Execution) *An execution E is a model of the state of a program at one moment in time and is defined as $E = (P, V, O, <)$, where*

» *P is the set of all processes;*

» *V is the set of all shared variables, i.e. (memory) locations;*

» *O is the set of all issued operations; and*

» *The transitive binary relation $<$ is a partial order on O.*

Among other details that will be explained further on, table 5.1 lists all operations. Reads and writes of a memory location in V are atomic. In practical systems, usually only reads and writes of bytes are indivisible and thus atomic. Handling variables that span multiple bytes is covered in section 5.3. The table also lists *patterns*, which are used to select operations with specific properties.

Definition 5.2 (Pattern) *A pattern, denoted $⦅\text{operation}, p, v, \text{value}⦆$, where $p \in P$ and $v \in V$, is a subset of O that matches any $o \in O$ that has the specified properties. A $*$ matches all.*

So, the pattern $⦅\text{w}, *, v, *⦆$ matches all writes to location v by any process, for example. Equivalent to w for a write, table 5.1 lists mnemonics for all other operations. Next, the initial state of a program is defined as:

Definition 5.3 (Initialization) *An execution $E = (P, V, O, <)$ is initialized, such that P contains all processes, V contains all locations, and $<$ is empty. All locations*

Table 5.1 – Orderings between existing and new operations on location v by process p

			new operation				
		pattern	r	w	R	A	F
previous operation	read	$\langle\!\langle \mathsf{r}, p, v, * \rangle\!\rangle$	$\prec_L$	$\prec_L$	$\prec_L$		$\prec_L$
	write	$\langle\!\langle \mathsf{w}, p, v, * \rangle\!\rangle$	$\prec_L$	$\prec_P$	$\prec_P$		$\prec_L$
	acquire	$\langle\!\langle \mathsf{A}, p, v, * \rangle\!\rangle$	$\prec_L$	$\prec_P$	$\prec_P$		$\prec_F$
	release	$\langle\!\langle \mathsf{R}, p, v, * \rangle\!\rangle$				$\prec_S{}^{\dagger}$	$\prec_F$
	fence	$\langle\!\langle \mathsf{F}, p, *, * \rangle\!\rangle$			$\prec_F$	$\prec_F$	$\prec_F$

† An acquire has its ordering $\prec_S$ on $\langle\!\langle \mathsf{R}, *, v, * \rangle\!\rangle$, not just on releases of the same process.

have an initial operation that behaves like a write and release. So, O is initialized, such that $\forall v \in V : |\langle\!\langle \{\mathsf{w}, \mathsf{R}\}, \epsilon, v, \bot \rangle\!\rangle| = 1$, *where ϵ is equivalent to all processes.*

Definition 5.3 states that all locations have an initial operation that is both a write and release. As a result, reads and acquires always have a predecessor.

5.2.2 Operations by processes

A program issues operations to the memory system. All operations that can be executed by any process are listed below.

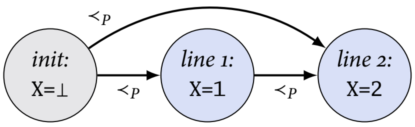
- *read*: retrieves the value of a previously executed write operation of a specific location.
- *write*: replaces the value of a location. Writes do not have to be visible for all processes immediately.
- *acquire*: gets an exclusive lock on a specific location. An acquire must be followed by a release of the same process. Moreover, mutual exclusion between an acquire and release is guaranteed by the platform.
- *release*: gives up the exclusive lock on a specific location.
- *fence*: adds dependencies to locally executed operations.

The properties of the operations are more formally discussed in section 5.2.4. When operations are executed, they add orderings to the execution graph that is being constructed.

Definition 5.4 (State transition) *When an operation o on location $v \in V$ by process $p \in P$ is executed, the next execution is $E' = (P, V, O', <')$, where $O' = O \cup \{o\}$, and $<'$ extends $<$ such that the ordering rules as indicated in table 5.1 apply to all matching operation patterns and o.*

Process 1:

```
X = 1;
X = 2;
```



Listing 5.5 – Program order of two writes

Without explaining those 'ordering rules' at this point, table 5.1 defines the rules that are applied between operations. For example, when a new write operation is executed, it will add the orderings $\prec_L$ between all previously executed reads on the same location by the same process and the new write, and it will similarly add the orderings $\prec_P$ between all previous writes and acquires and the new write. Therefore, the dependency graph grows by every new operation, and these orderings are never removed. The next subsection discusses these different types of orderings in the table in more detail.

5.2.3 Orderings: semantics of operations

Listing 5.5 shows a simple program with one process that executes two writes to the same location X. The graph shows that when X=1 is executed, one dependency is added from the initial write. This is graphically presented as $\text{A} \xrightarrow{\prec_*} \text{B}$, which indicates that every process observes that A occurred before B, because of the indicated ordering rule, where $\prec_*$ stands for some specific rule. When X=2 is executed, a dependency is added from all previous writes to the new one. We will omit the (implicit) initial write in the figures. Moreover, the figures are transitively reduced; all redundant orderings are left out of the figures, like the one from the initial write to X=2. The rule in this example is the *program order*.

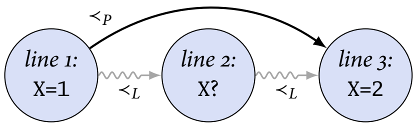
Definition 5.5 (Program order) *Program orderings $\prec_P$ are globally visible orderings between two operations of one process on one location.*

Definition 5.5 implies that writes of one process to different locations can be observed in a different order by different observers. Every process observes writes to the same location by one process in the same order, but the effect of the write does not have to be visible instantaneously.

A read will add ordering constraints that are only visible to the *local*, i.e. executing, process. Listing 5.6 gives an example. In this case, there is a relation between X=1 and the consecutive read; the compiler or hardware should not reorder these two operations. As a result, the read can only return the value 1. To determine the value read by a read, one can follow all global dependencies and dependencies that are local to the executing process, in reverse direction until a write is encountered (which is X=1 at line 1 in the example). This last-written value will be properly defined later on.

Process 1:

```
1 X = 1;
2 if(X==1)
3     X = 2;
```



Listing 5.6 – Local order of a read

Definition 5.6 (Local order) *Locally visible orderings $\overset{p}{<}_L$ are only visible to the executing process p.*

Graphically, a local ordering is denoted $\text{A} \overset{<_L}{\rightsquigarrow} \text{B}$, where only the executing process observes A occurring before B. All other processes could disagree. With this order, all local control dependencies in the program are preserved. The reads, writes, local order, and program order, as discussed so far, are equivalent to Slow Consistency.

Because the program order $<_P$ only orders per process, operations of two processes accessing the same location can be interleaved in any way. For inter-process orderings, synchronization is added. Synchronization consists of two operations: *acquire* and *release*, which behave in a mutual-exclusive way.

Definition 5.7 (Synchronization order) *Synchronization orderings $<_S$ are globally visible, per location orderings that can span multiple processes.*

Listing 5.7 on the next page shows a program with two processes that both try to acquire the same location. Depending on which process will get the lock first, process 1 reads either $\perp$ or 2. The figure shows how different ordering rules of table 5.1 on page 87 are applied.

Until now, it is impossible to enforce orderings between two locations. However, a communication pattern as of listing 5.8 on the next page is very common, where data in X is communicated by setting a flag, and another process waits until it receives the flag before reading the data. For that, a *fence*[3] is needed, which is similar to the fence, i.e. memory barrier, instruction of modern processors that force completion of earlier memory operations before later operations are executed.

Definition 5.8 (Fence order) *Fence orderings $<_F$ are globally visible, per process orderings that can span multiple locations.*

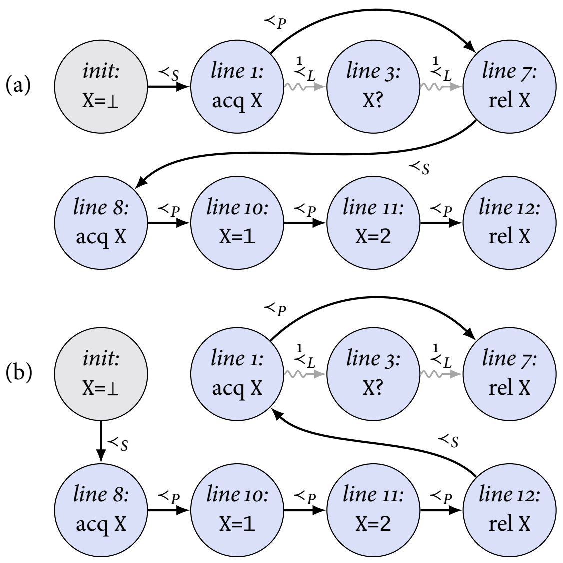
[3]The fences discussed in this section are applied on all locations at once. Without loss of generality, one could offer more complex fences on specific locations for optimization purposes. We do not discuss such kind of fences, as it complicates PMC's memory model abstraction too much.

Process 1:

```
1  acquire(X);
2  // critical section
3  r = X;
4  // r is either:
5  // (a)  ⊥, or
6  // (b)  2
7  release(X);
```

Process 2:

```
8  acquire(X);
9  // critical section
10 X = 1;
11 X = 2;
12 release(X);
```



Listing 5.7 – Exclusive access with two processes with a dependency graph for both possible interleavings. Regardless of which interleaving happens at run time, every observer agrees on that interleaving.

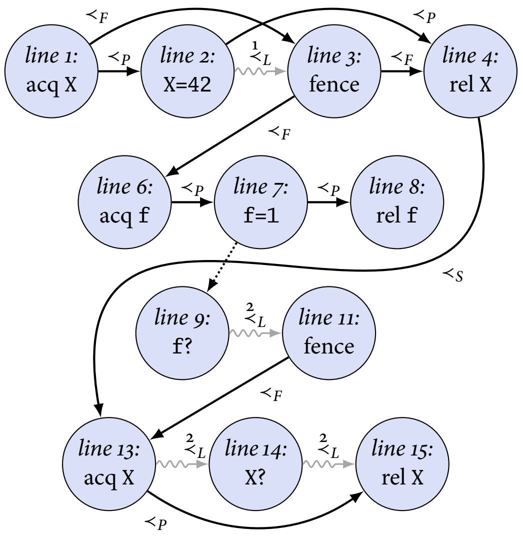
Initially: f=0

Process 1:

```
1  acquire(X);
2  X = 42;
3  fence();
4  release(X);
5
6  acquire(f);
7  f = 1;
8  release(f);
```

Process 2:

```
9  while(f!=1)
10     sleep();
11 fence();
12
13 acquire(X);
14 print(X);
15 release(X);
```



Listing 5.8 – Simple multi-core communication example

The fence of line 3 makes sure that `f` will be written after process 1 got the lock on `X`. The fence of line 11 prevents the compiler from moving the acquire at line 13 to before the while loop, where it (potentially) can acquire the lock before `X` is written. The dotted arrow indicates that when `f` is eventually observed being 1, it can be concluded that write of 1 must have been executed before. Although none of the ordering rules enforce it, this control dependency is valid, but only locally known to process 2. When process 2 acquires `X` afterwards, the fences make sure that it will always acquire after process 1 has acquired (and released) it. Therefore, it is guaranteed that process 2 will read the value 42.

Note that there is no way for process 2 to make sure the value 42 of `X` is read at line 14 without acquiring it; then there is no chain of dependencies that lead to the write of 42. Let us consider what happens when specific annotations would be removed. The acquire of `f` on line 6 is required, because there is no rule that enforces ordering between the fence on line 3 and a successive write to `f`. Similar, without the acquire of `X` on line 13, the read of `X` would not have to be executed after the fence of line 11. Moreover, without that acquire, there is no path from the read of `X` back to the write of 42; it might read the initial value of `X` as well.

Finally:

Definition 5.9 (Global order) *The globally visible ordering $<_G$ on two operations $a, b \in O$ is defined such that $a <_G b$ iff $a <_P b$, $a <_S b$, or $a <_F b$. All processes always agree on the orderings of $<_G$, no matter how the effects of the orderings are observed.*

Definition 5.10 (Execution order) *The execution ordering $<$ on operations $a, b \in O$ is defined such that $a < b$ iff $a <_G b$ or $a <_L b$. So, $<$ is a partial order on the operations O of an execution.*

Because now processes can have different views on the orderings, the point of view is included in the ordering relation. For two operations $a, c \in O$, we use the shorthand notation $a < c$ for describing $a <_G c$—the local orderings are not included, as the notation does not indicate the point of view. Additionally, $a \overset{p}{<} c$ says that a sequence of operations can be found between a and c that are either ordered via $<_G$ or $\overset{p}{<}_L$. In other words, this relation can recursively be defined as: $a \overset{p}{<} c$ iff $\exists b \in O : a <_G b$ or $a \overset{p}{<}_L b$, and $b \overset{p}{\leq} c$. Intuitively, $b \overset{p}{\leq} c$ is shorthand notation for $b \overset{p}{<} c \vee b = c$.

5.2.4 Observing slowly

Based on the ordering rules above, various properties of the operations can be defined more precisely. The last write operation of a location is the one that you first encounter when following the dependency graph in reversed direction.

Definition 5.11 (Last write) *The last write to $v \in V$ before operation $c \in (\!(*, p, v, *)\!)$ is denoted $W_c = \{a \in (\!(\mathrm{w}, *, v, *)\!) \mid a \overset{p}{<} c \land \nexists b \in (\!(\mathrm{w}, *, v, *)\!) : a \overset{p}{<} b \overset{p}{<} c\}$.*

W cannot be empty, because the initial write is included at least. If W contains multiple writes, reading the location is non-deterministic; a data race occurred. This leads to the conclusion that for a deterministic application, all writes to a single location must be in total order. As table 5.1 on page 87 shows, ordering between writes to the same location of two processes is only possible via acquires and releases, since $<_S$ is the only ordering that spans multiple processes. Therefore, all writes must be enclosed by an acquire and release—but a single acquire–release pair might contain multiple writes.

Definition 5.12 (Read value) *A read operation $r \in O$ by process p from location v returns either the last written value before r, or any value written afterwards. So, r can read $\{\mathrm{value}(b) \mid b \in (\!(\mathrm{w}, *, v, *)\!) \land \forall a \in W_r : a \overset{p}{\leq} b\}$, where W_r is the last write to r. However, when two read operations $r \overset{p}{<} r'$ by the same process read the same location, written by operations w and w', respectively, then this implies $w \overset{p}{\leq} w'$.*

So, a read can return an already overwritten value, because writes slowly propagate through the system. However, it is impossible to return an older value when previously a newer value has been returned. A formal description of such an observer function is given by Frigo [43].

In listing 5.8 on page 90, process 2 polls the flag. However, there is no control over when the write of process 1 arrives at process 2. It makes sense that a platform provides a *flush* function that makes writes globally visible sooner, but because the flush cannot be used to guarantee ordering, this is more a convenience; it is not part of the memory model.

5.2.5 Comparison to existing models

As stated above, the orderings and behavior of the read and write operations of PMC are identical to Slow Consistency. In literature, the globally observable orderings $<_G$, as defined by definition 5.9, are defined similarly but named differently:

1. $<_P$, combined with $<_S$, results in an ordering per location that spans multiple processes, which is equivalent to global data order *(GDO)*, as defined by Steinke and Nutt [101]; and
2. $<_F$ is an ordering per process that spans multiple locations, which is equivalent to global process order *(GPO)* [101].

For most synchronized relaxed models, Slow Consistency is assumed for reads and writes, and then different flavors of synchronization are added. When the writes to shared variables are wrapped in an acquire–release pair—which is necessary in order to be data-race free—the writes to a single location are in total order. As a result, the behavior is identical to Cache Consistency; a total order of writes per location and 'slow reads', where values propagate slowly through the systems. However, just having CC is not enough to implement the communication in listing 5.8 on page 90; fences are required. If one would add a fence between every operation, the model is equivalent to PC; a total order of all writes per location (GDO) and total order of all writes per process (GPO).

We argue that it is highly desirable that the platform supports both GPO (i.e. fences) and GDO (i.e. acquire–release pairs). Without GDO, which is the case for PRAM, non-deterministic execution cannot be confined and writing applications becomes extremely hard [43]. However, without GPO, it is not possible to simulate Sequential Consistency [113]. Relaxing the total order requirement of GDO to a partial order is proposed by Gao and Sarkar [44], but any implementation of it will be stronger [43]. So, both GDO and GPO are required to be usable, which is precisely what our model is based on.

Because it is possible in our model to apply all ordering constraints required to behave like PC, our model can benefit of all properties of PC, such as that it is able to simulate SC for data-race-free programs [6, 45]. However, our model allows specifying only the essential orderings, where PC overly constrains the possible orderings. Compared to Entry Consistency, our model is weaker, because of two additional relaxations:

1. acquire–release pairs of different locations by the same process are not ordered, unless a fence is applied; and
2. exclusive access (between acquires and releases) is allowed concurrently to read-only access.

5.3 Annotation and abstraction

Ideally, the PMC memory model, as discussed in section 5.2, should be the native model of a programming language, and the semantics of that language should only define orderings of the model. In that case, the language's syntax can help programmers to specify all required orderings, such that fewer errors are made. For now, such a language does not exist, so we introduce annotations that can be used in (existing) C programs. Adding ordering information by means of annotations is essential in the PMC approach.

5.3.1 Front-end: annotations in source code

Accesses to non-shared objects do not have to be annotated. As stated before, all writes to shared objects should be wrapped in an acquire–release pair to prevent data races[4]. To obtain symmetry in access pairs, *all* reads and writes should

[4]In this chapter, we assume that data races should be prevented. Chapter 6 will discuss that this does not necessarily have to be the case. In section 6.4, the set of annotations is extended to support specific data races.

Initially: f=0

Process 1:

```
entry_x(X);
X = 42;
fence();
exit_x(X);

entry_x(f);
f = 1;
flush(f);
exit_x(f);
```

Process 2:

```
do{
    entry_ro(f);
    poll = f;
    exit_ro(f);
}while(poll!=1);
fence();

entry_x(X);
r = X;
exit_x(X);
```

Listing 5.9 – Properly annotated source code of listing 5.8 on page 90

be wrapped, in either an entry–exit pair with exclusive read–write access or non-exclusive read-only access, similar to the acquire–release pairs of EC. Together with reads and writes, the annotations below cover all operations of table 5.1 on page 87.

» `entry_x(X)`: Issues an acquire operation on X. An `entry_x()` should be paired with an `exit_x()`.

» `exit_x(X)`: Issues a release operation on X. During an `exit_x()`, all writes to X do not necessarily have to be notified to others. An implementation could do a 'lazy release', which keeps all modifications to X local, until another process does an acquire of X. An eager release implementation would do a `flush(X)` (see below) before giving up the lock on X.

» `entry_ro(X)`: Marks the start of non-exclusive read-only access to X. In the implementation of this call, the system could take some effort to retrieve updates of X. An `entry_ro()` should be paired with an `exit_ro()`.

» `exit_ro(X)`: Marks the end of read-only access to X.

» `fence()`: Issues a fence operation. This should also prevent the compiler from reordering code and issuing proper fence instructions for an out-of-order processor.

» `flush(X)`: Because an `exit_x(X)` is lazy, a flush of X forces modifications to X to become globally visible. Concurrent read-only accesses then can receive the update. This is a best-effort operation, so there are no guarantees that all processes actually observe the modifications within a specific amount of time. It is only allowed to flush an object within `entry_x()` and `exit_x()`.

When these annotations are properly applied to the example of listing 5.8 on page 90, the resulting source code is shown in listing 5.9. The `flush(f)` is added to make sure that process 2 will read the value 1 eventually. A flush of X is not needed, because the acquire of X will always get the latest modifications.

The annotations are applied to shared objects of any size, which conflicts with the memory model. Recall, the memory model of section 5.2 assumes operations on variables of atomic locations, which must be just one byte. Most real-life data structures are larger than that, like a `struct` or a `double` on a 32-bit machine. In general, when such a multi-byte object is read, it is required that one protects the object with a mutex to prevent reading the new first half of the double and the old second half, for example. Hence, the compiler that processes the annotations must decide whether locking is required for read-only access. Although this decision is easy, it influences efficiency of the program.

With annotations in place (either by the programmer or a compiler), all information about the essential ordering of the application is available. Using this information, it is possible to map the application to the platform at hand.

5.3.2 Back-end example: three views on Starburst

Given the annotations of above, we claim that it is possible to map the application to any common multiprocessor hardware architecture, regardless of its supported memory model. For a sequential consistent system, the implementation of the annotations is trivial; mutual exclusion is still required for the entry–exit pairs, but all other annotations can safely be ignored, because the SC hardware already takes care of it.

We study the implementation of the annotations for hardware that implements a weaker memory model. For this, we use the 32-core Starburst system. This architecture is used to demonstrate three different memory systems:

1. a software-cache-coherent multiprocessor system (and the direct core-to-core ring and local memories inside the tiles are not used for data);
2. a DSM architecture, where all local memories are kept coherent via the ring, such that they form a shared memory (and the SDRAM is not used for inter-core communication); and
3. a setup where the local memory is used as SPM to hold a copy of the data that primarily resides in SDRAM (and the ring is not used for data).

At first glance, it seems non-trivial to use these three completely different architectures as back-end of the same memory model. However, the implementation of the annotations for these architectures is listed in table 5.2 on the following page and will be discussed below. For the experiments, we designed a single C++ interface that defines the annotations, where the implementation, i.e. back-end, can be changed transparently to the application.

The first setup relies on properly flushing the caches. The cache of the MicroBlaze is only capable to either invalidate dirty data in the cache, or flush dirty data and invalidate it afterwards. So, it is not possible to reconcile a dirty cache line, without also removing it from the cache. All shared objects are aligned to a cache line by compiler directives and cannot overlap with other objects. The second column of table 5.2 describes how the annotations are implemented for software cache coherency. This protocol resembles the Backer cache coherency protocol [19].

Table 5.2 – Implementation of PMC annotations on different memory architectures

annotation	*Software cache coherency*	*DSM over write-only interconnect*	*SPM and SDRAM*
read/write	By design, the MicroBlaze implements (at least) Slow Consistency. It exhibits in-order execution, and no interconnect reorders operations of one processor. So $<_L$ and $<_P$ between reads and writes are satisfied by the hardware.		
`fence`	Because the MicroBlaze is in-order, the fence only controls reordering by the compiler, and does not emit any instructions. So $<_L$ and $<_F$ between fences and other operations are satisfied by the hardware.		
`entry_x`	Exclusive access is enforced by acquiring a lock on a mutex that is related to the object that is protected. $<_S$ is implemented using the distributed lock (see section 4.5). To ensure $<_P$ between the acquire and successive operations, when the lock is transferred to another process…		
	…the object is flushed from the cache. So, the object does not reside in the cache outside of any entry–exit pair.	…the local version of the object is written to the local memory of the acquiring process.	…the acquiring process makes a local copy of the object's version in the SDRAM.
`exit_x`	Releases the lock on the object. Because the MicroBlaze is in-order, $<_P$ between the release and preceding operations is automatically guaranteed by the hardware.		
			First, the data is copied back to SDRAM.
`entry_ro`	When the object is `const` or its size is one byte, it does nothing. Otherwise, it acquires a lock on the object such that concurrent access by *`entry_x()`* is prevented.		Makes a local copy of the object. If the object is larger than one byte, the object is locked before copying and unlocked afterwards.
`exit_ro`	Flushes the corresponding cache lines and releases the lock if *`entry_ro()`* locked it.	Releases the lock if *`entry_ro()`* locked it, otherwise does nothing.	Discards the local copy.
`flush`	Flushes the corresponding cache lines.	Makes a copy of the object in the local memory to all other local memories.	Copies the object back to SDRAM.

In the DSM setup, the software must write local updates of the data to another's local memory via the (write-only) ring. When this is done properly, all local memories hold the same data and the MicroBlazes see the local memory as one single shared memory. The third column of table 5.2 shows the implementation to achieve this. Although reading each other's local memory is impossible, this shows that write-only access is sufficient to make memories coherent.

Finally, the SPM setup makes a local copy of the SDRAM for local processing. When the application is finished using the data, it is either copied back to main memory or discarded, depending on whether the data has changed. Although SPMs often require compiler support for higher efficiency, we chose to manage it at run time, because of simplicity of the implementation.

A single C++ interface might sound to introduce a lot of overhead. However, templates allow compile-time analysis of the types and operations on them, which lead to a highly optimized implementation. Let us discuss the implementation of software cache coherency on the MicroBlaze. Listing 5.10 on the next page shows a test program that wraps accesses to different types of variables in an *`entry_ro`*`()`–*`exit_ro`*`()` scope within the function `test()`. Three types of objects are tested: a constant `int`, a normal `int`, and a 64-bit `long long` (`int`).

An outline of the implementation of the entry and exit annotations is also presented in the listing. In conformance to table 5.2, *`entry_ro`*`()` checks the `const`ness and size of the object `o`, and calls (*br*anches to) `lock()` when appropriate. Assume that `lock()` (and its counterpart `unlock()`) take a lock on a mutex that is associated to the object. *`exit_ro`*`()` behaves similarly, but *`flush`*`()`es the object first from the data cache. Flushing the data cache can be done on a per-cache line basis. In contrast to x86, the MicroBlaze requires that all word-sized variables are aligned to the size of a word. Therefore, *`flush`*`()` checks whether the object is of such a type. Then, it either flushes the specific cache line the object's word resides in, or iterates over the memory range of the object, as it might overlap multiple cache lines. Checking the type is implemented using typical C++ partial template specialization trickery, which are completely evaluated at compile time. Therefore, these type checks are eliminated from the resulting binary.

To show the actual introduced overhead of the annotations, listing 5.11 on page 99 lists the MicroBlaze assembly output of the compiler for the three different data types. The listing shows the three `test()` implementations side-by-side. The left-most function, which corresponds to line 49 of listing 5.10, shows that no assembly is generated for *`entry_ro`*`()`. Only flushing the object by *`exit_ro`*`()` leads to one instruction: a `wdc.flush` (*w*rite to *d*ata *c*ache; *flush* the line to memory) of the memory address in register `r19`, which contains the address of `a`.

The second version of `test()` does lock and unlock the object `b`, as its contents are not constant. As a result, the assembly output is similar to the implementation of `test(a)`, but adds a function call to `lock()` and `unlock()` on lines 41 and 49. Finally, the right-most assembly output corresponds to `test(c)`. This version also locks the object, but flushing the object takes somewhat more effort. As the data

```
template <typename T> struct type_is_const          { enum{value=false}; };
template <typename T> struct type_is_const<T const>   { enum{value=true};  };

template <typename T> struct type_is_ptr            { enum{value=false}; };
template <typename T> struct type_is_ptr<T*>        { enum{value=true};  };
template <typename T> struct type_is_ptr<T* const>  { enum{value=true};  };

template <typename T1,typename T2> struct type_is_like{ enum{value=false}; };
template <typename T> struct type_is_like<T,T>       { enum{value=true};  };
template <typename T> struct type_is_like<const T,T> { enum{value=true};  };
// ...volatile qualifier support omitted

#define type_is_word(o)                               \
    (   type_is_like<o,char>::value              ||   \
        type_is_like<o,short>::value             ||   \
        type_is_like<o,int>::value               ||   \
        type_is_like<o,long>::value              ||   \
        type_is_like<o,unsigned int>::value      ||   \
        ...
        type_is_like<o,float>::value             ||   \
        type_is_ptr<o>::value                         \
    )

// Implementation of several annotations
template <typename T> void flush(T& o){
    if(!type_is_word(typeof(o)))
        do_flush_dcache_line(&o);
    else
        do_flush_dcache_range(&o,sizeof(o));
}
template <typename T> void entry_ro(T& o){
    if(!type_is_const<typeof(o)>::value && sizeof(o)>1)
        lock(&o);
}
template <typename T> void exit_ro(T& o){
    flush(o);
    if(!type_is_const<typeof(o)>::value && sizeof(o)>1)
        unlock(&o);
}

// Test program
template <typename T> void test(T& o){
    entry_ro(o);
    printf("%d\n",(int)o);
    exit_ro(o);
}

int main(){
    int const a=1;      test(a);
    int b=2;            test(b);
    long long c=3;      test(c);
}
```

Listing 5.10 – Implementation outline of the annotations for software cache coherency

```
 1 // test(int const& a)
 2 // function prologue
 3 addik      r1, r1, -32
 4 swi        r19, r1, 28
 5 swi        r15, r1, 0
 6 addk       r19, r5, r0
 7
 8 // entry_ro
 9
10
11
12 // call printf()
13 ...
14
15 // exit_ro
16 wdc.flush r19, r0
17
18
19
20
21
22
23
24
25
26
27
28 // function epilogue
29 lwi        r15, r1, 0
30 lwi        r19, r1, 28
31 rtsd       r15, 8
32 addik      r1, r1, 32
33 // test(int& b)
34 // function prologue
35 addik      r1, r1, -32
36 swi        r19, r1, 28
37 swi        r15, r1, 0
38 addk       r19, r5, r0
39
40 // entry_ro
41 brlid      r15, lock
42 nop
43
44 // call printf()
45 ...
46
47 // exit_ro
48 wdc.flush r19, r0
49 brlid      r15, unlock
50 addk       r5, r19, r0
51
52
53
54
55
56
57
58
59
60 // function epilogue
61 lwi        r15, r1, 0
62 lwi        r19, r1, 28
63 rtsd       r15, 8
64 addik      r1, r1, 32
65 // test(long long& c)
66 // function prologue
67 addik      r1, r1, -32
68 swi        r19, r1, 28
69 swi        r15, r1, 0
70 addk       r19, r5, r0
71
72 // entry_ro
73 brlid      r15, lock
74 nop
75
76 // call printf()
77 ...
78
79 // exit_ro
80 andi       r3, r19, -32
81 brid       .loop
82 addik      r4, r19, 8
83 .flush:
84 wdc.flush r3, r0
85 addik      r3, r3, 32
86 .loop:
87 cmpu       r5, r4, r3
88 blti       r5, .flush
89 brlid      r15, unlock
90 addk       r5, r19, r0
91
92 // function epilogue
93 lwi        r15, r1, 0
94 lwi        r19, r1, 28
95 rtsd       r15, 8
96 addik      r1, r1, 32
```

Listing 5.11 – MicroBlaze assembly of software cache coherency annotations

type is `long long`, it requires two words, which might reside in different cache lines. Therefore, flushing the cache requires a loop that iterates over the object's memory range. This can be recognized in lines 83 to 88: the cache line corresponding to the memory address stored in register `r3` is flushed, as long `r3` does not reach the end of the memory region, using increments of the size of the cache line of 32 bytes—`blti` stands for *b*ranch when *l*ess *t*han.

These three versions of `test()` show that the binary is highly optimized. The overhead of flushing a cache line can be reduced to just the flush instruction itself for most primitive types. Locking, however, will require more time, as discussed in the previous chapter.

In retrospect, the PMC memory model allows *abstraction* of the memory model of the hardware. The different implementations, as discussed above, show how software *complements* the memory model of the hardware to deliver the required guarantees to the application. The next section discusses the implementation of applications on the PMC memory model, and are portable to any of the aforementioned three architectures.

5.4 Case study

As a case study, we implemented applications for PMC for the three architectures of the previous section to show the feasibility of the approach.

5.4.1 Software cache coherency: SPLASH-2 benchmark

The first case study maps applications to the 32-core MicroBlaze system and focuses on adding software cache coherency transparently. As discussed in section 2.5, hardware cache coherency is one of the important issues that limit scalability to many cores, because of the complexity of hardware cache coherency protocols. On the other hand, software cache coherency is often discarded as a viable alternative, as it requires a strongly disciplined programming approach. As a consequence, shared data is predestined to be uncached in such a system. In this experiment, the annotations of section 5.3.1 are applied to investigate the feasibility of software cache coherency.

For three applications from the SPLASH-2 benchmark set, namely `radiosity`, `raytrace`, and `volrend`, two experiments are run:

1. A setup where all private data (the stack, heap, and data structures of the OS) is cached, but all application data that is shared between processes, resides in uncached memory. Therefore, no cache coherency protocol is required, and all cache flushes are nullified.
2. A setup where all memory is cached. Therefore, the protocol discussed above is applied on all shared data structures.

Figure 5.2 shows the performance results of both experiments, labeled 'uncached' for the first experiment with uncached shared data, and 'SW-CC' for the second. For all applications, it is indicated which percentage of the total execution time is used for the actual calculations, or the processor stalls. The stalls are categorized as: a stall on instruction cache miss; a stall on reading shared data (after a data cache miss or just an uncached read, depending on experiment); a stall because of a data cache miss when reading private data; and a stall on writing (hardly visible in the figure). For example, `radiosity` without cache coherency has an effective utilization of 38 %. Applying software cache coherency improves the total execution time by 26 % and the core utilization increased to 70 %. So, the execution time improved by 22 % on average for these applications when using software cache coherency, compared to leaving shared data uncached. The time spent on executing flush instructions for software cache coherency is for the three applications 0.66 %, 0.00 %, and 0.01 % of the total run time—the overhead is negligible.

The implemented cache protocol forces shared data out of the cache during the `*exit_x*()` and `*exit_ro*()` calls. So, executing two consecutive non-exclusive sec-

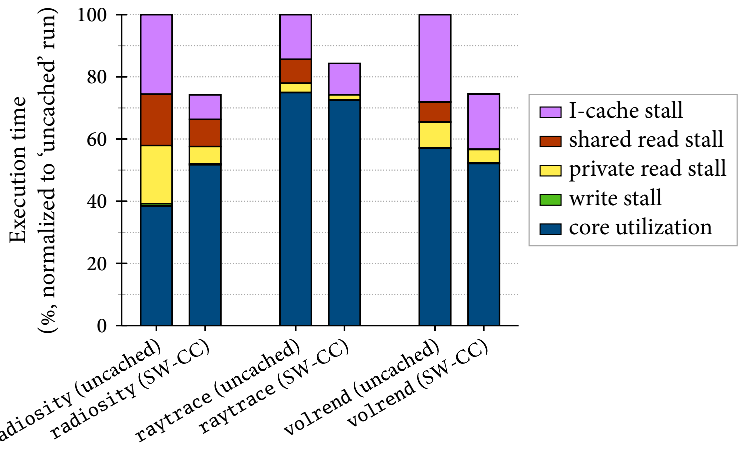


Figure 5.2 – Measured execution time and processor utilization of uncached and software cache coherency

tions will read data from background memory twice, even though this is strictly not necessary. Worst case, data is flushed from the cache after every read. In figure 5.2, the stall time on reading data is separated in reading private and shared data, of which the latter is conservatively (i.e. over-estimated) measured. The figure shows that for `raytrace` and `volrend`, there are hardly any stalls on reading shared data when applying software cache coherency. For `radiosity`, the stall time is reduced, although not as much as for the other applications. This is due to the design of the application, which addresses and updates the memory in a chaotic way.

Equivalent hardware cache coherency

Although the performance improved with software cache coherency, the overhead of this approach is important. A comparison of hardware and software cache coherency schemes is done by Adve et al. [5], which is based on compile-time analysis of memory operations using analytical models. In contrast, we have run-time measurements of the cache behavior, which allows a more realistic comparison.

Since we do not have a 32-core hardware cache coherent MicroBlaze system, we reason about the performance of such a system as follows. In the implemented protocol, shared data is flushed from the cache at an *`exit_x`*`()` and *`exit_ro`*`()` call. The platform can count the number of executed `wdc.flush` MicroBlaze cache-flush instructions, which precisely indicates the amount of flushed data. Two assumptions are made. First, the whole line that is flushed, contains valid data. So, multiplying the number of instructions by the cache line size gives an upper bound on the actual amount of data. Second, in a perfect hardware cache coherency implementation, all of this data is instantly communicated to all caches and is available to all cores.

Table 5.3 – Comparison measured software cache coherency overhead, and conservatively estimated maximum hardware cache coherency speedup

	Utilization[a] (%)	Flush instructions[ab] (%)	Read stall[ac] (%)	HW-CC speedup[d] (%)
`radiosity`	69.69	0.66	11.75	12.41
`raytrace`	85.89	0.00	0.03	0.03
`volrend`	69.96	0.01	0.16	0.17

[a] Measurement of the SW-CC runs of figure 5.2 on the previous page.
[b] Time the processor is executing instructions to flush the cache.
[c] Maximum time the processor stalls on reading shared data.
[d] Estimation, assuming that every cache read is a hit.

Hence, all read stalls on shared data are prevented, as data is always available in the cache. However, any realistic hardware implementation must be slower than this.

Based on these assumptions, a bound can be calculated how fast hardware cache coherency could be. The first column of table 5.3 lists the utilization of the processors for all applications, which corresponds to the utilization of the 'SW-CC' run of figure 5.2. This includes the time (see second column) required to execute the flush instructions, which is overhead of software cache coherency. For all applications, the time spent on flushing is very low, so the software overhead is very low. In the third column, the stall time on shared data is listed.

In case of hardware cache coherency, no time is required to flush the cache and no time is lost on stalling when reading data, because all data is assumed to be in the cache automatically. Hence, a hardware cache coherency implementation would benefit from the reduction of both. The sum of both numbers is the bound of the maximum improvement of having hardware instead of software cache coherency, which is also listed in the fourth column of the table. For example, `radiosity` uses 0.66 % of the time is used for flushes and stalls 11.75 % of the time on shared data. Therefore, the maximum reduction of the execution time is 12.41 %. Because the other applications share less data, the maximum speedup when hardware cache coherency would be available, is next to nothing. This shows that the maximum speedup by using hardware cache coherency is limited, but also very depending on (the design and implementation of) the application.

5.4.2 Distributed shared memory: multi-reader/-writer FIFO

The second case study uses the setup where all local memories are used as a single software-managed distributed shared memory system, which are all connected via a write-only interconnect. Although the SPLASH-2 applications above could be mapped onto this memory architecture, the local memories in our system are too small to put all data in them. Therefore, we discuss another application: a multiple-reader, multiple-writer FIFO. Such a FIFO, in combination with distributed memory, is useful in streaming applications [15, 40].

Listing 5.12 on the following page shows an outline of the implementation of the FIFO. For simplicity, only `push()` and `pop()` are given and checks for an `int` overflow of the pointers `write_ptr` and `read_ptr` have been left out. The listing indicates which ordering rules apply to the statements in the source code. A nice property of this implementation is that the read and write pointers are only polled from local memory, which is fast and does not influence the execution of other processors. The DSM back-end (see table 5.2 on page 96, third column) makes sure that updates will arrive properly.

Although this example is given in the context of distributed memory, the FIFO behaves also correctly on all of the other architectures.

5.4.3 Scratchpad memory: motion estimation

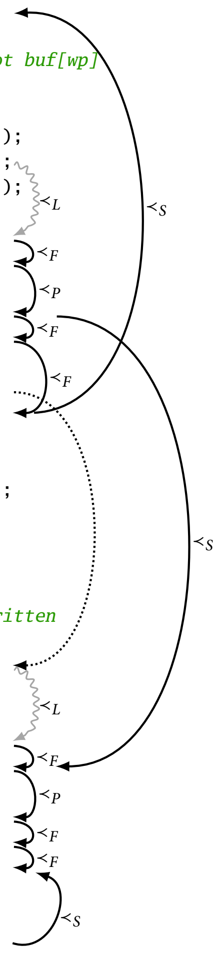
The last case study shows how the PMC approach can be used for a typical SPM application: motion estimation. In video encoding, the motion of an object is used for compression. For this, a video frame is split in a matrix of blocks. Then, every block of the next frame is matched within a search window of a reference frame. A naive algorithm to find the motion vector is to do a full search. In such an approach, it is efficient to store both the block and the search window locally, because they are read many times. In that context, an SPM can be beneficial.

There is a practical issue when dealing with an SPM when the processor does have an MMU: an object has two addresses, one of the main memory and one of the SPM. It is convenient when the annotations hide this. We implemented several C++ classes, as an example of how such complexities can be hidden and how dealing with the memory model is better integrated in the language.

Listing 5.13 on page 105 gives a partial C++ implementation of a motion estimation application and the annotations for SPMs. Assume that the `worker()` function is executed by one thread, which gets work packets via a queue. Then, it accesses the search window and block, and executes the matching function to determine the motion vector. The *`entry_ro`*`()`–*`exit_ro`*`()` calls are handled by the `ScopeRO` class, where the entry call is implemented by the constructor and exit by the destructor. The implementation corresponds to the fourth column of table 5.2. For the entry–exit pair with write access, there is a similar class, but this is not shown in the listing. When the `ScopeRO<Window>` object is cast to a `Window` **`const`**`&` as a function parameter on line 30 in order to access the actual data, for example, a reference is returned to the SPM and the original data is left untouched. Although the concept of the annotations stays the same, this shows that it depends on the language how they can be used effectively.

Like the previous examples, the application is now independent of the underlying memory model. Although it depends on many architectural parameters, experiments show a significant performance increase when this application is using SPMs, compared to the software cache coherency setup.

```cpp
template <typename T,int N,int R> class MFifo {
    T buf[N];
    int write_ptr, read_ptr[R];
public:
    void push(T data){
        int wp,rp;
        entry_x(write_ptr);
        wp =    write_ptr % N;
        // Wait until all readers got buf[wp]
        for(int i=0;i<R;i++)
            do{
                entry_ro(read_ptr[i]);
                rp =     read_ptr[i];
                exit_ro( read_ptr[i]);
            }while(rp<wp-N);
        fence();
        entry_x(buf[wp]);
                buf[wp] = data;
        exit_x( buf[wp]);
        fence();
                write_ptr++;
        flush(  write_ptr);
        exit_x( write_ptr);
    }
    T const pop(){
        int wp,rp,me=get_reader_id();
        entry_ro(read_ptr[me]);
        rp =     read_ptr[me] % N;
        exit_ro( read_ptr[me]);
        do{
            // Wait until data is written
            entry_ro(write_ptr);
            wp =     write_ptr;
            exit_ro( write_ptr);
        }while(wp<=rp);
        fence();
        entry_x( buf[rp]);
        T data = buf[rp];
        exit_x(  buf[rp]);
        fence();
        entry_x( read_ptr[me]);
                 read_ptr[me]++;
        flush(   read_ptr[me]);
        exit_x(  read_ptr[me]);
        return data;
    }
};
```



Listing 5.12 – Outline of a multiple-reader, multiple-writer FIFO in C++, with element type T, a buffer depth of N, and R readers. The essential orderings are indicated.

```
// implementation of annotations (see table 5.2 on page 96)
template <typename T> class ScopeRO {
    T const & obj;
    T* spm;
public:
    ScopeRO(T const & o) : obj(o) {   // entry_ro
        spm = (T*)alloc_spm(sizeof(T));
        if(sizeof(T)>1) lock(obj);
        memcpy(spm,&obj,sizeof(T));
        if(sizeof(T)>1) unlock(obj);
    }
    ~ScopeRO { free_spm(spm); }       // exit_ro
    operator T const &() { return *spm; }
};

// application code
typedef struct {
    Window const * window;
    MBlock const * mblock;
    Vector* vector; } work_t;

Vector motion_est(Window const &,MBlock const &);

void worker(){
    work_t work;
    while((work=queue.pop())){
        ScopeRO<Window> window_s(*work.window);
        ScopeRO<MBlock> mblock_s(*work.mblock);
        ScopeX<Vector>  vector_s(*work.vector);
        vector_s = motion_est(window_s,mblock_s);
        // all scope objects destructed
    }
}
```

Listing 5.13 – More complex scoping support in C++, with an alternative approach to handle entry–exit pairs

5.5 Conclusion

Porting applications to a platform with a different programming model requires intrusive modifications to that application. A change in the memory model is a commonly encountered issue. This chapter presents PMC, an approach that makes applications independent of the memory model of the hardware, in order to allow transparent mapping to different platforms. This effectively removes the hardware's memory model from the programming model.

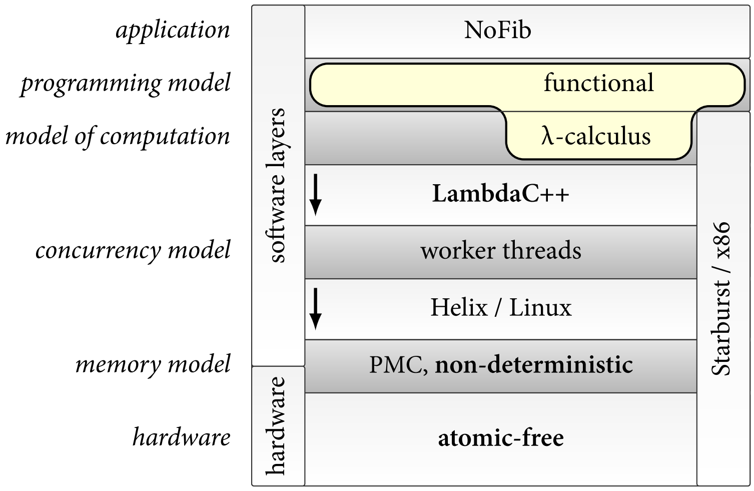
PMC consists of a weak synchronized memory model that defines the fundamental orderings an application can assume, and annotations that allow defining additional ordering constraints. The memory model 1) is an intersection of all orderings of all common memory models to allow maximum ordering flexibility; but 2) is still strong enough to behave like Processor Consistency, and can therefore simulate SC for data-race-free applications [45]; 3) is weaker than Entry Consistency, because of relaxed constraints on the ordering of synchronization operations; and 4) clearly distinguished the four different types of orderings, which allows straightforward usage. The annotations in the application give the tooling all information about the additional ordering requirements, such that it can automatically insert logic to complement the hardware orderings when necessary.

The overview figure on page 76 suggests that the memory model is removed from the programming model. That is not entirely true for C; the programmer still has to annotate the source code. However, these annotations are related to the algorithm that is implemented, and are not related to the actual hardware. In that sense, specifying annotations is required by the abstract machine within the programming model. The case study in section 5.4.3 shows that using C++, the annotations can be embedded in the language quite well, such that PMC is completely hidden. To prevent even deliberately casting pointers to circumvent these scope classes, a programming language is required that natively uses PMC and automatically wraps all accesses in proper entry–exit pairs.

The case studies show the interplay of hardware and software, which are combined in three different ways to realize the same memory model with different trade-offs: hardware vs. software control over cache coherency, core-to-core vs. via main memory communication, repeatedly reading main memory vs. overhead of duplication and reading local. The software cache coherency case is especially interesting, as a common approach to programming a multiprocessor system is to use C, shared memory, and caches. The evaluation shows that, depending on the application, using hardware cache coherency might only give a comparable performance to using software cache coherency, but requires complex hardware control. Therefore, abstracting from the memory model, as PMC does, gives great freedom in the implementation of the platform, and the stable interface to the applications allows them to be portable between platforms.

This empty page leaves some room for random thoughts:

What is 'smart' about a smartphone when it lacks the fundamental property of intelligence, namely an understanding of the environment? It never knows what I want; I always have to tell it what to do next…



Chapter 6 overview

6

Implicit Software Concurrency

Abstract – *Previous chapters assumed programming a many-core system using an imperative language. Then, atomicity is preferred to reason about the program state, by means of atomic RMW operations, a strong memory model, and hardware cache coherency. This chapter shows the impact on the platform when a λ-calculus-based (functional) language is used instead. Ordering requirements of memory operations are more relaxed and synchronization is simplified, because λ-calculus does not have a notion of state or memory, and therefore imposes fewer ordering requirements on the platform. We implemented a functional language for architectures with a weak memory model, without the need of hardware cache coherency, any atomic RMW operation, or mutex—in other words, the execution is* atomic-free. *Moreover, both the memory model and concurrency model can be hidden from the programmer, as the programming paradigm implicitly allows concurrency.*

The relation between a platform and its programming model is an interesting one. Where chapter 3 presents the coherence of models and hardware, chapter 5 presents the influence of hardware on the programming model. Trends show that the memory model becomes weaker, so programming has to become more disciplined. To overcome the programming difficulties, the previous chapter introduced an abstraction, such that the memory model of the hardware could be removed from the programming model.

Based on the hardware trends above and the influence of these trends on the programming model, one might conclude that the hardware drives modifications of the programming model in general. For example, architectural choices for many-core systems clearly lead to changes in programming models. Let us consider the effects of the interconnect more closely. Where two cores can share one bus, larger

Large parts of this chapter have been published in [JHR:7].

systems often have complex interconnection structures like a network-on-chip, which increase the latency of the communication between cores. This complicates operations that require atomic global communication, where the result of such an operation must become visible to all cores, without any (observable) intermediate state during the state transition. A single write operation in hardware with a strong memory model is one example of such atomic global communication. Additionally, a hardware cache coherency protocol and an atomic RMW operation, like a compare-and-swap, are also complex and time-consuming to realize in hardware when global communication takes multiple clock cycles to complete. Alternatives that avoid atomics, and are therefore easier to realize in hardware, are software cache coherency, or not to use cache coherency at all, and to drop atomic RMW operations. However, these alternatives complicate programming. When state changes are not instant anymore, and thus need some time to complete, the transient state is unpredictable, but still observable in a multicore environment. The hardware and common programming models often expose these issues to the programmer, which makes reasoning about correct program behavior very hard [4]. Hence, the choice of the (interconnect) hardware affects programming.

However, this only partly addresses the hardware–software relation. The problems mentioned above are all related to memory consistency and synchronization, or concurrency in general. A widely used concurrent programming paradigm to harness the power of a parallel machine is threading in combination with shared memory. However, Lee [71] argues that threads (in combination with an imperative language like C) induce non-determinism, which all should be pruned away by the programmer. Having a strong memory model and efficient synchronization makes this task a bit easier, but also makes the hardware more complex and less scalable, which leads to the problems above. Hence, one might conclude that the choice of a *programming* paradigm drives the design choices regarding the *hardware*. This is in contrast to what one might have concluded above. Where previous chapters modified hardware abstractions and analyzed the effects on software, this chapter modifies the programming abstraction and analyzes the effects on hardware.

In all previous chapters, we assumed that the platform is programmed using C or C++. In this chapter, we show that the requirements for a multicore architecture relax, when assuming that it is programmed using a functional language. More specifically:

1. We show that concurrent execution can be achieved without locks and atomic RMW operations, even on hardware with a weak memory model, based on properties of the programming paradigm. Hence, the execution is *atomic-free*; it does not rely on any sequence of operations that should be observed by or communicated to other processes atomically, in either hardware or software. This opens the possibility to reduce hardware complexity, and therefore makes the hardware more scalable. We acknowledge that avoiding *all* atomics is a very strong requirement, and that practical systems might benefit from allowing some of them anyway. However, we

show that it is possible to do so, and still provide a proper programming interface.

2. We show that carefully introducing data races in the run-time system does *not* harm the deterministic behavior of the application, which is in contrast to data races in both C11 and C++11 standards.
3. We derive rules for a weak memory model, and show the relation to PMC and its software cache coherency back-end specifically.
4. Experiments on Starburst and x86 show the feasibility of the approach.

This can be achieved, because of the nature of the λ-calculus, which is the mathematical basis of the functional programming paradigm. Its distinctive properties include that it does not have the notion of state or memory, which eases dealing with weak memory models. Moreover, functional programs naturally allow concurrency, because all dependencies between calculations are explicitly defined. Furthermore, since a functional language is single-assignment, calculating the same expression twice gives the same result, which allows simplification of synchronizing concurrent calculations.

We discuss the basic idea of our approach next, followed by related work, a discussion of our functional language that allows atomic-free execution, the requirements on the memory model, and experiments.

6.1 Basic idea

The basic idea of this approach is exemplified as follows. Consider the following pseudo-code:

```
x = foo();
y = bar() + x;
```

If this code snippet was C, the assignments of x and y should be done in the specified order, otherwise the initial value of x is used for the addition instead. When both lines are calculated by different threads, the computation of y by one thread should be stalled until it is guaranteed that the other thread finished computing x.

In a functional language, variables are single-assignment, so '=' means *definition* instead. One can imagine that the thread calculating y checks whether x has been computed yet, and if not, it waits for the completion of x or computes x by itself. Hence, a data race exists in the last case in the calculation and assignment of x. However, even if x is evaluated twice because of this data race, the result is the same.

Allowing this data race can be used for optimizations, without influencing the outcome of the program. Threads might decide to compute variables repeatedly to prevent fetching it from shared memory and consuming precious memory bandwidth. Additionally, distributing work among worker threads without proper synchronization might also (safely) result in duplicates. Moreover, communication of results to other cores can be postponed when that seems to be beneficial for cache coherency protocols, for example.

However, there is no free lunch. In contrast to C, it is not obvious to decide whether a variable is still in use or not, as the source code does not define when a variable is not used anymore. Keeping administration at run time is possible, but data races might complicate this analysis. So, there is a balance in allowing races and arbitrary ordering of memory operations during evaluation for higher performance on one hand, and preventing races and giving guarantees about the memory state for garbage analysis on the other.

6.2 Related work

Many functional languages exist, and they handle concurrency (and the related problems) differently. Clo*j*ure [32] runs in the Java VM and assumes worker threads on top of a shared-memory machine. SAC is based on a fork–join approach [48]. Haskell supports different flavors of parallelism, based on GHC: annotations and implicit concurrency [77], explicit threads and channels [92], and data parallelism [28]. All implementations assume that the application is executed on an SMP machine, with a POSIX-like OS, which implies having a strong memory model and threads. Ports of Haskell to other architectures include House [57] and GHC's port to ARM, but they do not support multiple cores. We focus on the fundamental requirements of executing a parallel functional program, instead of assuming a common architecture. To the best of our knowledge, no work focuses on the direct relation between these languages and an underlying hardware architecture with a weak memory model, for example.

Other parallel functional languages are based on message passing, like Erlang [10], Eden [76] and Multi-MLton [98]. These languages can be ported to many architectures, because the message-passing abstraction hides issues related to memory consistency, and the model fits nicely to networks of computers. The same holds for stream processing applications that are implemented using message passing. However, sending and receiving messages has overhead by (unnecessary) data duplication, and it enforces a specific form of synchronization. As shared memory is more generic [35], and we focus on concurrency within a single system-on-chip, we do not consider message passing.

In a different direction, the functional programming paradigm can also influence processor design instead of the system architecture. PilGRIM [21] is an example of a specialized processor for lazy languages. The authors propose a multicore system as future work. However, it likely encounters the same memory consistency issues as any other multicore system, as the memory layout of expressions during execution is similar to the one discussed in section 6.3.2.

Although not specifically for functional languages, Bhattacharjee et al. [13] measured the overhead of synchronization primitives of the parallelization libraries

OpenMP and Intel's TBB. Even though the authors propose optimizations to improve the measured synchronization overhead of respectively 47 % and 80 % of the benchmark runtime, they conclude that the overhead will remain high at higher core counts. In contrast, we eliminate the need for such synchronization primitives, by choosing a different programming paradigm than the one of C/C++.

Avoiding locks and atomic instructions has also been proposed by Tithi et al. [108] for breadth-first search algorithms. These operations are recognized as costly, and the experiments with their proposed solution outperform state-of-the-art algorithms. Nasre et al. [84] also conclude that RMW operations are costly and discuss transformations of graph algorithms to eliminate them, specifically targeting GPUs. These techniques make modifications to the algorithm, where we avoid atomic operations in general at the level of the programming paradigm. Optimizing the algorithm itself is beneficial, and it is an orthogonal technique to the modifications to the platform.

On larger scale systems like cluster computers, MapReduce [38] is a popular approach to program for concurrency. In this model, a function is concurrently applied to every element of a large dataset (*map*), and the individual results are combined into a smaller dataset, like the sum of the inputs (*reduce*). Because both phases do not modify the dataset, they can be considered side-effect free, which in turn can tolerate processing node failures. Work that does not complete in time can easily be reissued to another node in that case. Although MapReduce assumes 'embarrassingly' parallel applications and large datasets on disks, the benefits also apply to functional languages at a smaller scale, like a single multicore system. However, functional languages are more generic, as they do not assume such a specific form of parallelism in the application.

6.3 Shift in paradigm: λ-calculus and its implementation

To understand the execution and memory related issues of programs written in a functional language, we (informally) explain the fundamentals of such a language. Afterwards, the implementation of our functional language is discussed, which closely follows these fundamentals.

6.3.1 Background on λ-calculus

Functional languages are based on their counterpart in mathematical logic, λ-calculus. This formal system defines expressions or *λ-terms* as

$$M ::= c \mid x \mid M\ M \mid \lambda x.M$$

where c can be any constant or primitive function, x is a variable that binds a name to another λ-term, $M\ M$ is an application of the second λ-term to the first one, and $\lambda x.M$ is a function that takes one argument and binds it to all occurrences of x in M. Prefix notation is used for function application. For example, $(\lambda x.((+\ 2)\ x))$, or just written $\lambda x. + 2\ x$, is a function that takes one argument and adds 2 to it. (Although the + operator does not exist in λ-calculus, assume that its behavior is defined.)

In this simple example, + is (assumed to be) a function that takes two arguments. If only one argument is applied to it, like $(+\ 2)$, the result is still a function, but now requires one argument—supplying fewer arguments than the function requires is *partial application*. Functions can also be used as arguments. Functions that can take and/or return functions are *higher-order functions*. An example is function composition, $f \circ g$, which is defined as $C = \lambda f.\lambda g.\lambda x.f\ (g\ x)$. When both f and g are applied to C, the result is a function that still requires one argument.

Next, the only rule of computation is called *β-reduction*, which substitutes a formal argument by an actual one. So, $(\lambda x. +\ x\ 2)\ 3$ is reduced to + 3 2, which then can be computed as 5.

The order in which expressions should be reduced is not defined. For example, the expression $(\lambda x.f\ x)\ ((\lambda y.g\ y)\ 7)$ can be reduced to both $(\lambda x.f\ x)\ (g\ 7)$ and $f\ ((\lambda y.g\ y)\ 7)$ in the first reduction step. However, based on the Church-Rosser Theorem [31], the fully reduced result is always the same—in this case $f\ (g\ 7)$. Evaluating a function is *side-effect free*; it only computes a result, and does not change the system in any other way. Therefore, reducing a term can also be postponed until its value is required, which is exactly what a *lazy* functional language does.

A more program-like example is the following definition of the volume of a cylinder with radius 2 and length 5:

$$\begin{aligned} \text{main} &= \text{cylinder } 2\ 5 \\ \text{cylinder} &= \lambda r. \times (\times\ \pi\ (\text{sqr } r)) \\ \text{sqr} &= \lambda x. \times\ x\ x \end{aligned}$$

When we repeatedly reduce main, the result is computed:

$$\begin{array}{ll} \text{main} & \rightarrow_{\text{substitute}} \\ \text{cylinder } 2\ 5 & \rightarrow_{\text{substitute}} \\ (\lambda r. \times (\times\ \pi\ (\text{sqr } r)))\ 2\ 5 & \rightarrow_{\beta} \\ (\times\ (\times\ \pi\ (\text{sqr } 2)))\ 5 & \rightarrow_{\text{substitute}} \\ (\times\ (\times\ \pi\ ((\lambda x. \times\ x\ x)\ 2)))\ 5 & \rightarrow_{\beta} \\ (\times\ (\times\ \pi\ (\times\ 2\ 2)))\ 5 & \rightarrow \\ (\times\ (\times\ \pi\ 4))\ 5 & \rightarrow \\ 62.83\ldots & \end{array}$$

We implemented a simple functional language that closely follows the definition of λ-calculus and the β-reduction rule, which is discussed next.

6.3.2 Our simple functional language: LambdaC++

Based on the definition of λ-calculus, it does not fundamentally require hardware features such as a strong memory model, and fully deterministic execution. As a proof-of-concept to show that it is possible to realize an atomic-free execution of a concurrent program, we implemented an untyped functional language[1]. In fact, the language is just C++, where λ-terms are represented by functors—classes that overload the `()`-operator, such that the syntax resembles λ-calculus somewhat and functions can be used as function arguments. We will refer to this language and its implementation as *LambdaC++*.

We will discuss the aspects of the implementation where usually atomics are involved. Notably, the focus is on data races during β-reductions, and the distribution of work via a work queue.

General setup

To explain the actual execution of a functional language, let us introduce a simpler example, of which every step in the computation will be discussed. This program increments 5 two times:

$$\begin{aligned} \text{main} &= \text{inc}\ (\text{inc}\ 5) \\ \text{inc} &= \lambda x . + x\ 1 \end{aligned}$$

This shows several statically allocated functions and objects, namely main, inc, +, 5, and 1. The reduction steps until the result is a constant are as follows. In these steps, the argument substitution is done immediately when a function is replaced by its definition.

$$\begin{aligned} 1:&\quad \text{main} && \rightarrow \\ 2:&\quad \text{inc}\ (\text{inc}\ 5) && \rightarrow \\ 3:&\quad +\ (\text{inc}\ 5)\ 1 && \rightarrow \\ 4:&\quad +\ (+\ 5\ 1)\ 1 && \rightarrow \\ 5:&\quad +\ 6\ 1 && \rightarrow \\ 6:&\quad 7 \end{aligned}$$

Figure 6.1 on the next page shows for every step above, how it is executed on a computer. In this system, the static objects are constant, stored in memory, and always accessible. The stack is the same stack as used for executing C programs; it holds all local variables, function return addresses, etc. However, only the relevant pointers to terms are depicted in the figure. The stack is always initialized to contain a pointer to main. The heap is used to allocate λ-terms and constants. In the figure, the heap and stack grow upwards.

[1] The implementation is available under the GPLv3 license at `https://sites.google.com/site/jochemrutgers/lambdacpp`.

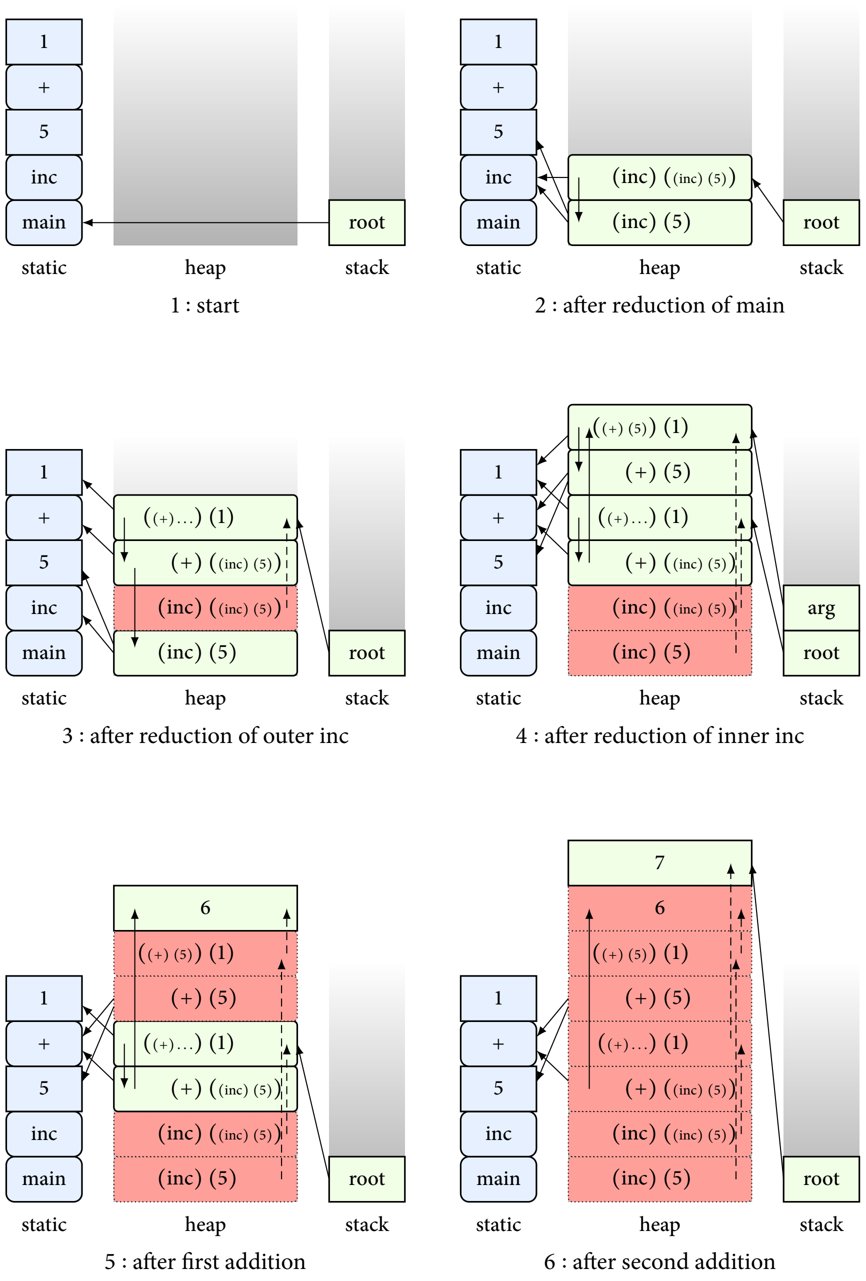


Figure 6.1 – Example of evaluation of LambdaC++

Step 1 starts with an empty heap and a single pointer on the stack, which points to main. The system repeatedly applies the reduction rule on the top-most pointer on the stack. After the first reduction step, which substitutes main, the state of the system is also depicted in the figure. The expression inc (inc 5) is split into two applications. Every application consists of a pointer to one function and a pointer to one argument. Step 2 in the figure shows that the heap contains these two λ-terms: one application that points to inc and 5, and another application that points to inc and the result of the first application. The arrows indicate the objects on which the heap objects depend. The 'root' element on the stack now points to the result of the reduction, which is the outer application of inc. Since this result is not a constant, the process is repeated.

In step 3, the outer inc function is reduced to + (inc 5) 1. In this case, the function + requires two arguments, namely inc 5 and 1. Since an application can only apply one argument to a function, *currying* is used. Currying will create a chain of applications, which all apply one argument to the previous expression. Therefore, the first application applies the existing expression inc 5 to the function +, which results in a partially applied function that still requires one argument. Then, the second application applies 1 to this partial function application. Again, the root on the stack now points to the reduction result. Additionally, the figure indicates the relation between a λ-term and its reduction result by a dashed arrow. This means that every term that points to inc (inc 5), should follow this indirection. Moreover, the application inc (inc 5) is now unreachable by following pointers starting from the stack, so this term is *dead*. In a later stage, its memory will be freed.

Next, + (inc 5) 1 should be reduced. Assume that it is known that + can only add constants. So, both arguments must be reduced to constants first, before the actual addition can be done. Although this system uses lazy evaluation, this is the moment where actual data is required, so reductions of the arguments are forced. To this extent, the system pushes the non-constant argument inc 5 on the stack, which is reduced to + 5 1. This is done in step 4. Hence, there can be multiple pointers on the stack, which point to terms that are currently being reduced. After full reduction, the term inc 5 is eventually replaced by the constant 6 in step 5. The argument of the +, which the root pointer points to, is now fully reduced to a constant and therefore popped from the stack. Now, the actual addition can be done.

Finally, step 6 creates the constant 7 on the heap, which is the result of the program. Now, all other terms are unreachable, hence dead, but still occupy heap space. Later, we will discuss the garbage collector that will free this memory.

A simplified implementation of the (program-independent) C++ classes is shown in listing 6.1 on the following page. Among many other details, handling of superfluous function arguments and partial function application are left out. Listing 6.2 shows two programs. Because g++ is used, optimizations are only applied at the C++ level; g++ is oblivious of the functional properties of the program. Although adapting GHC and using Haskell is possible, modifying the fundamentals of such a large system is practically not feasible within the time constraints of our project.

```
class Term {                            // generic lambda-term super class
    Term& indirected;
public:
    Term& operator()(int c){
        return *new Application(*this,*new Constant(c));}
    Term& operator()(Term& t){
        return *new Application(*this,t);}
    virtual Term& Reduce(){return *this;}
    virtual Term& Evaluate(Term& args...);
    virtual void SetIndirection(Term& to){indirected=to;}
};

class Constant : public Term {          // an (int) constant
    int i;
public:
    Constant(int i) : i(i) {};
    virtual void SetIndirection(Term& to){}
};

class Application : public Term {    // function-argument application
    Term &func,&arg;
public:
    Application(Term& func,Term& arg) : func(func), arg(arg) {};
    virtual Term& Reduce(){return SetIndirection(func.Evaluate(arg));}
    virtual Term& Evaluate(Term& args...){return func.Evaluate(arg,args...);}
};

class Function : public Term {          // a wrapper for a C++ function
public:
    Function(Term& (*func)(...));
    virtual Term& Evaluate(Term& args...){return func(args...);}
};

Function mult,add,par,pseq;             // some functions of a standard library
```

Listing 6.1 – Simplified C++ implementation of LambdaC++

```
// cylinder program
Term& sqr_func(Term& x){                return mult (x) (x); }
Function sqr(sqr_func);
Term& cylinder_func(Term& r){           return mult (mult (pi) (sqr (r))); }
Function cylinder(cylinder_func);
Term& main_func(){                      return cylinder (2) (5); }
Function main(main_func);

// parallel version of the double inc program
Term& inc_func(Term& x){                return add (x) (1); }
Function inc(inc_func);
Term& main_func(){                      Term& a = inc (5);
                                        return par (a) (inc (a)); }
Function main(main_func);
```

Listing 6.2 – Example programs in LambdaC++

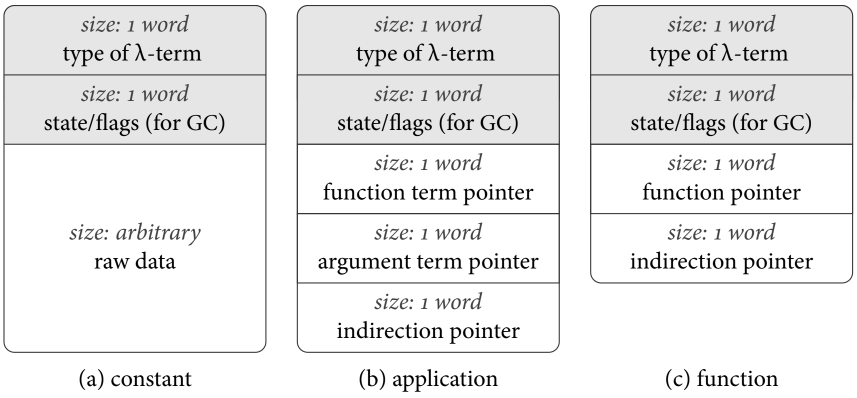


Figure 6.2 – Memory layout of λ-terms

The implementation supports integers, (complex) doubles, and arbitrary large numbers via the GNU MP library, although the language is in principle untyped. Therefore, the compiler does not check these types. Because C++03 does not allow anonymous functions, the λ-expression of the form $\lambda x\,.\,M$ should be lambda lifted [63], which means that they can only be defined as a named function, like sqr, cylinder, and inc in listing 6.2, and not occur somewhere inline.

Given the information required for λ-terms in general and the described approach to execute β-reductions, we derive a generic memory layout, which is depicted in figure 6.2. All terms have in common that they contain their type (the *vpointer*, in C++ lingo). Next, administrative fields for garbage collection *(GC)* are added, which depend on the specific GC approach. A constant only has to contain the raw data. An application requires a pointer to the argument term that is applied, and a pointer to the function term that the argument is applied to. (When a function requires multiple arguments, a chain of applications is used by means of currying.) A (named) function, like sqr, needs to store the function pointer, e.g., to `sqr_func`, to call upon computation.

Constants cannot be reduced any further. Since the size of the reduction result of the other types of λ-terms is unknown on beforehand, it is in general not possible to replace terms by their result in-place. Instead, new memory is allocated for the result, and an indirection pointer to that result is set. Therefore, functions without arguments and applications need to contain room for the indirection pointer to the eventual reduction result.

When this indirection pointer is set, the term is superseded. Then, the function and argument that are pointed to, are not considered to be required anymore and can be garbage collected eventually. Hence, all contents of the λ-terms as shown in figure 6.2 are constant after initialization of the term, except for the indirection pointer.

Worker threads and parallelism

Concurrency is exploited by running one worker thread per core, which concurrently reduces parts of the program. Each thread has its own heap that contains λ-terms. Parallelism is introduced by the par function, which is very similar to the one of Parallel Haskell. The programmer has to use the par function in order to run (parts of) the program in parallel; this is not done automatically. This function pushes one of its arguments on a work queue, allowing other workers to pick it up and start to eagerly reduce the term. A standard library defines par as follows:

$$\text{par} = \lambda x.\lambda y.y \qquad \text{, and } x \text{ is put on a work queue during } \beta\text{-reduction.}$$

The side-effect of par does not influence the outcome of the program; it only triggers the start of a parallel execution. The earlier example is modified to use par as follows:

$$\begin{aligned} \text{main} &= \text{par}\ a\ (\text{inc}\ a) \qquad \text{, where } a = \text{inc } 5; \\ \text{inc} &= \lambda x. + x\ 1 \end{aligned}$$

The outcome of the program is still 7. However, inc 5, which is labeled a, can be computed by another worker than the one that computes main. The first few reduction steps of this program are:

$$\begin{array}{lll} 1: & \text{main} & \rightarrow \\ 2: & \text{par (inc 5) (inc (inc 5))} & \rightarrow \\ 3: & \text{inc (inc 5)} & \rightarrow \\ 4: & + \text{ (inc 5) 1} & \ldots \end{array}$$

The corresponding states of the machine that executes these reduction steps, is depicted in figure 6.3. At step 2, the heap contains the application of the two arguments to par. The variable a is temporarily put on the stack while constructing the applications to par. This way, both references to inc 5 point to the same application, which prevents that the application is created and computed twice. This is necessary to ensure that once inc 5 is reduced, both uses of the term see this reduction result.

Next, at step 3, the first argument to par, which is inc 5, is pushed on the work queue, and the worker thread continues the evaluation with reducing inc (inc 5). In the figure, the double border indicates that the data is shared among threads. The work queue only contains a reference to inc 5, although it is not shown in the figure. Assume that another worker thread removes this reference from the work queue and starts evaluating it. Now, two worker threads are concurrently executing the application. This is depicted by the bottom two states of the figure: the bottom left state shows the initial worker that already executed the previous steps, the bottom right state is the other worker that just concurrently computed inc 5 to the constant 6. This saves the initial worker some time, as the addition of inc 5 and 1 can be executed immediately, in contrast to the evaluation steps in figure 6.1.

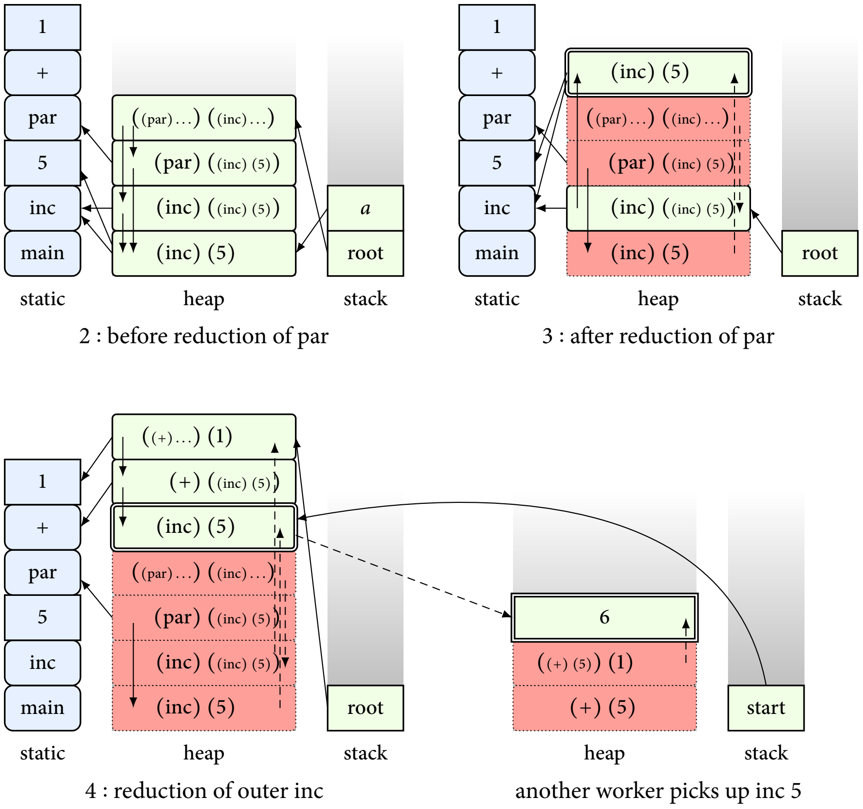

1
+
par
5
inc
main
static
heap
stack
a
root
2 : before reduction of par
3 : after reduction of par
4 : reduction of outer inc
6
start
another worker picks up inc 5

Figure 6.3 – Example of evaluation with par

So, par allows work to be computed concurrently, but it does not guarantee that it will be done that way. It is possible that all other workers are busy, and the work is never picked up from the queue, after which the original worker has to perform the task anyway. Moreover, there exists a race condition where two workers start computing the same term simultaneously. We will address this race condition in more detail later on.

Par is useful when a significant amount of work can be sent to another worker thread. Consider the following definition, which (inefficiently) calculates the Fibonacci number of a given index in the sequence.

$$
\begin{aligned}
\text{fib} &= \lambda n.1 && \text{, when } n \le 2;\\
\text{fib} &= \lambda n.+ \ (\text{fib}\ (-\ n\ 1))\ (\text{fib}\ (-\ n\ 2)) && \text{, otherwise;}\\
\text{pfib} &= \lambda n.\text{pseq}\ (\text{par}\ a\ b)\ (+\ a\ b) && \text{, where } a = \text{fib}\ (-\ n\ 1)\text{, and}\\
& && \qquad\quad b = \text{fib}\ (-\ n\ 2).
\end{aligned}
$$

Where fib calculates the Fibonacci number by one worker, pfib puts *a* on a work queue first to be computed in parallel, then calculates *b* itself, and adds them afterwards. This program uses pseq. This function breaks the normal lazy reduction order by forcing to compute the first argument before continuing with the rest. Similar to par, pseq is defined in a standard library:

pseq = $\lambda x . \lambda y . y$, and x will be fully reduced first upon β-reduction of pseq.

In practice, par is used as an annotation to indicate that the value of its argument is expected to be needed in the future, and that the programmer expects that its computation might take some time. Therefore, it is beneficial to start computing on par's argument in parallel, which is *a* in the example above. On the other hand, pseq is used to annotate that it is beneficial to compute its argument immediately, instead of lazily. In combination with par, this gives control over the distribution of workload over the workers. So, the functions par and pseq do not influence the outcome of the program, but are just hints how the program might be executed faster.

The combination of worker threads and par introduces the notion of local and shared data: all data is local, until it is applied to par. In order to share the data with other workers, the term is then made globally visible, which influences how it is determined whether a term is dead or not, and how data should be accessed regarding memory consistency. The implementation ensures that local terms can refer to both local and global, i.e. shared, ones, but global terms only refer to other global terms [7]. So, terms are either global or local and are always owned by the worker thread that owns the heap a term resides in.

As par hints that its argument requires a significant amount of work to compute, it is probably wise to make sure that this computation is done only once. To this end, when one worker evaluates the term and another one requires it meanwhile, the second worker should *stall* until the first one has finished computation. If the second worker would also start computing the term, compute power is wasted. So, par does the following:

1. Make sure that the term is globally visible, by duplicating it as a global term. Such a global term can reside in the same heap as the local term does, but the C++ class just handles accesses to it differently.
2. Add a *black hole* to prevent double work, by indicating that a term is 'under evaluation', such that other workers can wait for the result. In the implementation, a black hole is a subclass of `Term`. Upon reduction of the black hole, it eagerly reduces the term it protects, i.e. points to, and sets the black hole's indirection pointer to the result afterwards. When another worker tries to reduce the black hole, it stalls until the indirection pointer is written.
3. Put a reference to the black hole (and therefore the duplicated term) on a work queue.

4. Set the local term's indirection pointer to the newly created (black hole that protects the) equivalent global term.

We have seen above that a worker can be in a few different phases: idle when out of work, running β-reductions, stalling on a black hole, and doing garbage collection. Later on, we present experimental results regarding the time spent in these phases. Unlike GHC, when evaluation of a term blocks (for example on a term that is currently evaluated by another worker), the worker thread just blocks; no context switching has been implemented between evaluation of multiple (unrelated) terms.

Everything that happens at run time, such as doing β-reductions, handling of distribution of work among workers, allocation, and garbage collection of memory, are part of the run-time system *(RTS)*.

Local vs. global garbage collection

There exist multiple approaches to garbage collection *(GC)*. As the concepts of this chapter are independent of the chosen approach, we consider the approach and implementation of garbage collection less relevant. Therefore, we only briefly discuss a high-level overview of the approach that is used in LambdaC++. We chose to use a mark–sweep approach [64]. The algorithm works according to the following steps: 1) it marks all terms on its heap as dead; then 2) it marks all terms that are pointed to from the program stack, as alive; next 3) it follows pointers from living terms to other terms, until no new living term is found; and finally 4) all dead terms are freed. To find the root of the computation, the stacks of the worker threads must be inspected to find whether there are active references to terms. As this stack also contains other data, like function return pointers and (possibly outdated) register contents, properly finding these active references is hard. Therefore, we use a shadow stack [52], which tracks only the λ-term references on the normal C++ stack.

There are two flavors of GC:

» Local: Only local terms are cleaned from the heap. This can be done independent of other workers, because it is guaranteed that no local terms are used by other workers. All encountered shared, i.e. global, terms are assumed to be alive. Because only locally accessible terms are processed during local GC, memory consistency is irrelevant; no other worker reads or writes these terms.

» Global: Both local and global terms are cleaned from all heaps. This can only be done in a stop-the-world fashion, where all workers stop the current evaluation and participate in a GC run. The synchronization between workers can be done by a (Pthread) barrier. Although using such a barrier could be relatively costly when the implementation is based on polling memory, it does not influence the performance much, as this is a relatively rare operation—later on we present measurements, which indicate that β-reductions are done orders of magnitude more often. For example, a shared-

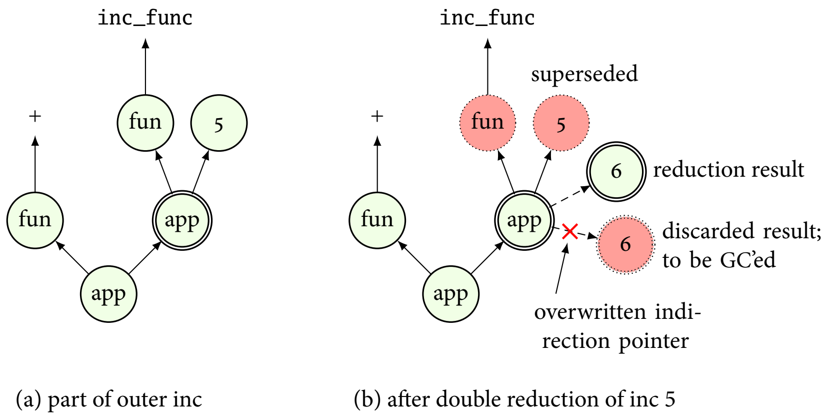

inc_func
+
fun
5
fun
app
app
(a) part of outer inc
inc_func
superseded
+
fun
5
6
reduction result
fun
app
6
discarded result; to be GC'ed
app
overwritten indi-rection pointer
(b) after double reduction of inc 5

Figure 6.4 – Steps in computation

> memory polling-based algorithm like the bakery lock [69] suffices. As we focus in this chapter on concurrency issues during evaluation, a discussion about the internals or optimization of the GC is beyond the scope of this work. Moreover, as the GC is written in 'normal' C++, it uses weak memory models in a general fashion, which has been covered chapter 5.

The local GC is invoked when the currently allocated heap memory is exhausted, which happens a dozen times per second. When not enough garbage is collected, more memory is requested from the OS. Global collection is invoked every second, but never in the midst of an arbitrary function; the RTS can only switch to GC when it is idle, or a new term has to be created and new memory is allocated. In contrast to interrupts, which can arrive at any time, the execution of the program and the system are therefore always in a known state.

6.3.3 The atomic-free core: data races and lossy work queue

As discussed section 6.3.1, computation in λ-calculus is done by repeatedly doing β-reduction on a term, until it results in a constant, or it is a partially applied function. Since a β-reduction is side-effect free, it is safe to allow some non-determinism.

Figure 6.4(a) exemplifies a part of the total graph during step 4 of the example with the double inc of section 6.3.2. When the application of inc and 5 is reduced, the result is 6, and the application term is indirected to this result. Figure 6.4(b) shows the graph when two workers have reduced this term at the same time. Both workers can update the indirection pointer, which results in a data race. However, it does not matter how others observe this update; the result is the same either way. During GC, the race is reconciled, and one result is properly discarded.

Setting the reduction result does not require locks, when we assume that writing the indirection pointer is atomic. However, there does not have to be a well-defined (total) order in writing this pointer. The fact that this race condition can safely be ignored, is a great potential for performance improvement by using a weak memory model, since synchronization requirements relax. Section 6.4 will define what is required to allow these races.

The argument why data races can be allowed during evaluation, also applies to the distribution of work: duplication of terms is not a problem, since the result is always the same. As mentioned before, a queue is used that is populated using par. GHC implements such a queue as a lock-free (work-stealing) FIFO queue. Its implementation does not lock a mutex, which otherwise might prevent other threads to progress when the thread that locked it, is context-switched or blocks on a shared resource, for example. However, a lock-free data structure is based on atomic RMW operations, such as a compare-and-swap [54]. These operations are hard to implement in hardware and, more importantly, not required for our queue.

We chose to design this queue as a *lossy stack*. The rationale behind the choice for a stack instead of a FIFO queue, is that newly pushed work onto the stack is more relevant to start computing on than older terms, as these older terms are more likely to be computed already by the thread that pushed it. The stack can be lossy, because it is allowed that race conditions prevent terms from being pushed at all, and that popped terms are popped twice at the same time. In the former case, the thread that pushed the work will compute the term by itself when required, in the latter case, the black hole will prevent doing the work twice.

Here is a trade-off between allowing incidental losses/duplicates over using locks or RMW operations. In systems with hardware support for RMW instructions, using them can be beneficial. However, we show that the lossy stack allows avoiding atomics, but still guarantees correct program behavior.

Pseudo-code of the implementation of the lossy stack is shown in listing 6.3 on the next page. The annotations `*fence*()` and `*flush*()` behave as defined in chapter 5. The writes to `m_queue` and `m_top` are not protected by a lock, so data races occur. The `*FENCE*()` annotation is similar to a normal fence, but differs at one crucial point. It does not only guarantee that operations before will be executed earlier than those after the fence, but it must also make sure that all writes to the specified object, e.g., `m_queue[top]` at line 11, *complete* before later writes. Hence, processes observe a *global order* between writes to the specified object before the `*FENCE*()`, and writes to *any* location after the `*FENCE*()`. Because this ordering is stricter than is defined for `*fence*()`, the function name is capitalized to express the difference. The next section relates this additional guarantee to the `*fence*()` annotation used in PMC.

During global GC, the contents of the lossy stack are also used as the roots of computation. Although there are several race conditions in the implementation of listing 6.3 on the following page, these are only relevant during evaluation of the program. It is safe to walk over the stack during global GC, because no worker modifies the stack at that time. Race conditions during evaluation are addressed in the next section.

```
class LossyStack {
    int volatile m_top;
    Term* volatile m_queue[SIZE];
public:
    LossyStack() : m_top() {}

    void push(Term* term){
        int top=m_top;
        if(top<SIZE){
            // write object
            m_queue[top]=term;
            // make sure pointer is
            //  written after term
            FENCE(m_queue[top]);
            // increment pointer
            m_top=top+1;
            flush(m_top);
        }
    }

    Term* pop(){
        int top=m_top;
        // make sure pointer is
        //  read before term
        fence();
        if(--top>=0){
            // read term
            Term* res=m_queue[top];
            // decrement pointer
            m_top=top;
            flush(m_top);
            return res;
        }else
            return NULL;
    }

};
```

Listing 6.3 – Lossy stack

6.4 Impact on memory consistency and synchronization

When a functional program is executed concurrently, multiple worker threads reduce terms at the same time and might even reduce the same term simultaneously. Section 6.3.3 showed that λ-terms are constant during their lifetime, except when the indirection pointer is set after reduction and the term becomes superseded. Based on this sequence, we can derive rules how the memory should behave such that races are allowed, but the program's result is deterministic.

This section relates these rules imposed by the programming paradigm to the memory model of the hardware. We will focus on operations on λ-terms, and then make the translation to operations on memory locations and PMC.

6.4.1 A λ-term's life and rules

Every λ-term has the following sequence of phases during its lifetime:

1. *Allocation on the worker's heap*
   A term is either local or global, depending on the context in which it is created. They can share the same heap, but accessing a global term requires attention regarding memory consistency, which is discussed in a moment.
2. *Initialization of the memory*
   In our case, the constructor of the C++ object takes care of this.

private access:

Only the owning thread accesses the λ-term. The term's content is constant after initialization.

3. *Indirecting another term to this one*
   A term is always a result of a reduction, so there exists a term that is replaced by the newly created one. Setting an indirection pointer to the new term will make it visible for other workers, which might follow the pointer.
4. *Replace the term by the result of a reduction*
   After β-reduction, the indirection pointer is set, which is the same operation as of phase 3, but from a different perspective. A race condition exists, because multiple workers might reduce the same term simultaneously.
5. *Term dies*
   When no pointer exists to this term, it becomes unused and can be garbage collected. Because the number of pointers to a term change at run-time, and it is subject to data races, this event is not detected during evaluation. Only during GC, the application graph is stable and can be analyzed.
6. *Deallocation during GC*
   At this point, the heap memory is freed.

shared access:

The term is valid and globally accessible. Only the indirection field can be overwritten by concurrent threads.

private access:

The owning thread destructs and cleans up the term.

From this list, we can identify all operations that can be executed on a term, namely: *construction* (phase 1 and 2); *read* (during phase 3 and 4), where a worker reads the term after following a pointer to it; *indirect* (phase 4), where a worker sets the indirection pointer to the reduction result; and *destruction* (phase 6). For these operations, we discuss which guarantees, i.e. rules, are necessary to be implemented by the platform. Such a guarantee is something a worker thread can assume to be always valid.

Although the intended behavior of the operations identified above is rather straightforward, the interaction between these rules is more complicated. In a similar manner as chapter 5 defined how reads and writes behave, the four operations on λ-terms have rules that define the required orderings to properly allow the execution of a functional program.

Construction of a term is obviously more than just a single read or write of memory. However, only the worker that creates the term can access this memory, because other workers do not have knowledge about its existence yet. So from a memory consistency point of view, this can be seen as a single operation. Any consecutive operation on the term should see the constructed term, which leads to the formulation of the following rule the memory subsystem must comply with:

Rule 6.1 (Construction) *Any worker that executes an operation on an existing term should observe that its construction has been completed.*

Although this sounds trivial, it means that the underlying system must make sure that the initialization of the term is completed and globally visible before a pointer to it is exposed to another worker. So, when another worker reads or sets the indirection pointer, the platform must make sure that the term's construction has been completed.

When an indirection pointer of a term is set, the following rule must apply:

Rule 6.2 (Indirect) *Setting the indirection pointer from term t_1 to term t_2 is atomic and in globally total order with respect to other operations on term t_1 by the same worker and the construction of t_2.*

The restriction that writes should be atomic is usually already fulfilled by hardware, because pointers have (usually) the size of one machine word. If that is not the case, writing such a pointer will have overhead by locking and unlocking the related memory location. As described in section 6.3.3, writing the indirection pointer twice does not harm the outcome of the program. Therefore, such writes do not have to be in total order, which is usually the case for memory models. The non-determinism by this data race is allowed, but should be solved during GC later on.

Next, workers can read a term, possibly multiple times.

Rule 6.3 (Read) *Reads of a term are in a total order with respect to other operations on the same term by the same worker.*

During GC, the program state is analyzed for dead terms. These terms should not be accessed afterwards.

Rule 6.4 (Garbage collection) *Before a worker destructs a term, all reads and indirections by any worker should be completed first.*

This also means that after destruction, no worker should read or indirect the term anymore—otherwise the garbage analysis was faulty. Because the state of the memory is fixed during GC, any non-determinism in the indirection pointers can be solved by completing all outstanding writes first. This results in a single state of the application, which every worker agrees on.

For every pair of executed operations, one of the four rules applies. Table 6.1 summarizes which rule applies for every pair of a previously executed operation and a new one.

6.4.2 Mapping from rules to PMC

For Sequential Consistency, the four rules of the previous section are trivial to guarantee; the memory model gives all guarantees already. For weaker models, specific countermeasures have to be taken to give the guarantees by the platform.

Table 6.1 – Rules that pairs of operations on λ-terms are subject to

previous operation			new operation c	i	r	d
	construction	c	×	6.1	6.1	6.1
	indirect	i	×	6.2	6.3	6.4
	read	r	×	6.2	6.3	6.4
	destruction	d	×	×	×	×

× Impossible

The λ-calculus-based programming model does not expose the memory model to the programmer. Therefore, an implementation of the language can use optimizations for the specific memory model of the hardware it is compiled for. However, the memory abstraction, as defined by PMC in chapter 5, is also useful to use in this case. One can imagine that the RTS of LambdaC++ is implemented in such a way, that it supports multiple target architectures. To be concrete, we run the same RTS on an Intel platform, as well on Starburst using software cache coherency. Therefore, the RTS is built upon the memory model and annotations of PMC to properly support both memory architectures at the same time. Note that the application programmer does not specify the annotations for PMC by hand, but the compiler and RTS insert them automatically.

A straightforward way of applying annotations that conform to all rules, is to wrap all operations inside an `entry_x()`–`exit_x()` pair, which guarantees exclusive access. In this way, the memory orderings are as strict as Sequential Consistency. However, using exclusive access defies the purpose of atomic-free execution, and it is stricter than necessary. Therefore, we define what is required to implement the rules in this context.

Unprotected write access

The first two rules, rules 6.1 and 6.2, define that a thread writes a term for either construction or indirection. According to PMC's annotation guidelines, every write should be wrapped in a scope with exclusive access. This is a reasonable approach, assuming that an application is programmed using an imperative language like C, and reasoning about state is a key feature of such a language. However, in Lambda-C++, it is guaranteed that no two workers can construct the same term at the same time, or data races in writes are allowed, so using a lock is unnecessary. Because of this change in assumptions, we define a new type of access scope in addition to exclusive (read/)write access and non-exclusive read-only access: *non-exclusive (read/)write access.*

Such a non-exclusive write access has exactly the same semantics as `entry_x()` and `exit_x()`, except that the underlying acquire after a release is ordered $<_P$ on

Table 6.2 – Orderings between operations on location, i.e. λ-term, v by process p, which are based on table 5.1 on page 87, and tailored to LambdaC++'s requirements

			new operation					
		pattern	r	w	R′	A′	F	F′
previous operation	read	$\llparenthesis \mathsf{r}, p, v, * \rrparenthesis$	$\prec_L$	$\prec_L$	$\prec_L$		$\prec_L$	$\prec_L$
	write	$\llparenthesis \mathsf{w}, p, v, * \rrparenthesis$	$\prec_L$	$\prec_P$	$\prec_P$		$\prec_L$	$\prec_P$
	non-exclusive acquire	$\llparenthesis \mathsf{A}', p, v, * \rrparenthesis$	$\prec_L$	$\prec_P$	$\prec_P$		$\prec_F$	$\prec_F$
	non-exclusive release	$\llparenthesis \mathsf{R}', p, v, * \rrparenthesis$				$\prec_P$	$\prec_F$	$\prec_F$
	fence	$\llparenthesis \mathsf{F}, p, *, * \rrparenthesis$			$\prec_F$	$\prec_F$	$\prec_F$	$\prec_F$
	strong fence	$\llparenthesis \mathsf{F}', p, *, * \rrparenthesis$		$\prec_F$	$\prec_F$	$\prec_F$	$\prec_F$	$\prec_F$

all releases of the same process on the same location, instead of the synchronization order $\prec_S$. Therefore, it does not take a mutual exclusive lock on the object. In a similar way as table 5.1 on page 87 is defined, the ordering of the non-exclusive acquire and release is defined by table 6.2. Let us use `entry_w()` and `exit_w()` as annotation for such a scope. It has an important consequence: concurrent `entry_w()`–`exit_w()` pairs lead to data races, or more specifically, the last write set W can have multiple elements.

Listing 6.4 shows pseudo-code of how operations on λ-terms should be annotated. The construction of the λ-term `t2` is wrapped in a non-exclusive write scope. At this point, no data races can occur, because no other process has knowledge of the existence of the term, so no process has a reference to that specific memory location. Setting the indirection pointer from `t1` to `t2` is also done with non-exclusive access. As multiple threads might do this simultaneously, a data race can occur, but this is not harmful, as discussed before. In the listing, process 2 receives the reference to `t2` and uses this term afterwards.

Communicating the term to other threads via setting an indirection pointer, resembles the example of listing 5.8 on page 90, where the variable `X` is communicated to others by setting a flag. For that particular example, we argued that `entry_x()` was require to read `X` at the receiver, because it was impossible to guarantee that the latest value of `X` was read otherwise. The difference between the `X` in that example and the term in listing 6.4 is that a λ-term is unknown to other processes upon construction. When other workers learn about a λ-term's existence, it initial value is that of the construction, in contrast to `X`'s $\perp$. Hence, where a lock for `X` was required, using locks for a term's construction is not. This shows that properties of the model of computation influence requirements of the memory model.

Stronger fence

To guarantee that construction is completed before any later operation on the term, the `FENCE()` comes into play, which was already introduced in section 6.3.3. As defined before, it adds a global (fence) order between writes to the specified object,

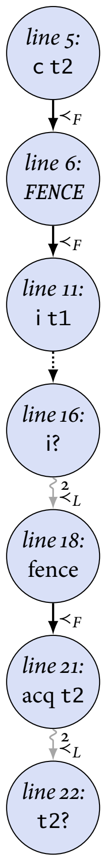
Initially: `Term t1` exists

Process 1:

```
 1  // construct
 2  Term* t2=t_alloc();// Allocate memory for term.
 3  entry_w(*t2);      // Only this process has a reference to
 4                     //  t2 so non-exclusive access is safe.
 5     new(t2) Term(); // Invoke t2's constructor.
 6     FENCE(*t2);     // Force completion.
 7  exit_w(*t2);
 8
 9  // indirect
10  entry_w(t1);       // Non-exclusive, possible data race.
11     t1.SetIndirection(*t2);
12     flush(t1);      // optional
13  exit_w(t1);
```

Process 2:

```
14  // Assume indirection is set.
15  entry_ro(t1);      // Poll indirection pointer of t1.
16     Term& t2=t1.GetIndirection();
17  exit_ro(t1);
18  fence();           // Force order of access to t1 and t2.
19
20  // read (use) term
21  entry_w(t2);       // Non-exclusive, possible data race.
22     t2.Reduce();    // ...or any other method using t2.
23  exit_w(t2);
```



LISTING 6.4 – Ordering dependencies of λ-term operations

the term pointed to by `t2` in this case, and any successive write. More precisely, the *FENCE*(`t`) is identical to *fence*(), but adds the ordering $<_P$ between earlier $(\!(\mathtt{w}, p, \mathtt{t}, *)\!)$ and the *FENCE*() operation, where p is the executing process, and adds the ordering $<_F$ between the *FENCE*() and all later writes by the same process. This strong fence is also listed in table 6.2. So, the *FENCE*() makes sure that the modifications to `t2` are written to memory, before continuing to set the indirection, in this case.

To use the reference to `t2` by process 2, *entry_w*() is used to access `t2`. In this case, *entry_ro*() cannot be used, as it does not enforce ordering between the read of the indirection pointer of `t1` and `t2`. On the other hand, *entry_x*() is too restrictive, as locking is not required. *entry_w*() does issue an acquire operation (although without locking), and therefore has a fence order $<_F$, which will lead to the chain of dependencies as visualized by listing 6.4.

The *FENCE*() is also an important feature to realize the garbage collection rule, rule 6.4. In the mark phase of GC analysis, a *FENCE*() should be used on the term that is marked. When the worker thread signals that is finished marking all active terms, the *FENCE*() makes sure that all outstanding writes will complete. Therefore, a worker thread either does not have a reference anymore to a (dead) term, or marks the term properly as being active.

PMC's extended back-end

We added two annotations to PMC that make executing λ-calculus-programs more efficiently. The implementation of the *entry_w*()-*exit_w*() pair in the back-end of PMC is identical to *entry_x*()-*exit_x*() pair, except that locking does not have to be done. Therefore, all accesses to λ-terms are done in an atomic-free manner.

Regarding *FENCE*(), hardware that supports Sequential Consistency, already complies with the requirements of *FENCE*(). For software cache coherency, as is used in Starburst, it will flush the corresponding cache lines, followed by a normal fence. As Starburst uses in-order processors and interconnects, cache flushes and posted writes will arrive at the memory in the same order as they are issued. In a distributed system, PMC's back-end has to take into account that later writes should not overtake earlier ones, which can be the case when an out-of-order NoC is used, or data is written to multiple memories with different write latencies.

The next section will present experimental results based on two systems with a different memory hierarchy, and therefore a different back-end for the memory model annotations.

6.5 Experiments

We tested LambdaC++ on two architectures. The first architecture is a hyper-threaded 12-core Intel Xeon system, which contains in this case 24 logical cores in total, and runs Linux. On this system, the scalability of our atomic-free execution and Haskell is tested. The second architecture is Starburst with 32 cores and Warpfield. Note that Starburst does not have atomic RMW operations, and uses software cache coherency.

The workload for the tests is delivered by applications from the parallel section of the Haskell NoFib Benchmark Suite [88] (see also section 2.8.3). We implemented five of them in LambdaC++, namely `coins`, `parfib`, `partak`, `prsa`, and `queens`, and will compare them to the Haskell versions in the experiments.

6.5.1 Scalability and speedup

All Haskell applications are compiled with GHC 7.4.2 for the Intel platform. Our functional language runs on both the Intel and the Starburst platform. During the experiment, the speedup of the applications is measured, depending on the number of cores used. Figure 6.5 on page 134 shows the results for all applications and platforms, which is the average of five runs with a standard deviation of the execution time that is below 6 %.

In the figure, the speedup is shown, which is the multicore performance relative to the sequential run. So, with *n* cores, i.e. worker threads, a speedup of *m* means that the wall-clock execution time is *m* times less when *n* cores are utilized in parallel, compared to the execution time on one core. Note the striking resemblance to figure 4.10 on page 73. The execution of LambdaC++ requires about 400 instructions on average per created λ-term, including allocation, β-reduction, and garbage collection. Still, the performance is about 100 times lower than that of a fully optimized Haskell implementation. We expect that this difference stems from the fact that GHC generates more efficient code, but also evaluates fewer λ-terms, as it is able to optimize the program at the functional level, which g++ is oblivious of. Even though the absolute performance differs, the speedup shows similar behavior on x86. Both the Haskell and LambdaC++ versions show a close-to-linear speedup for about the first ten cores[2]. After that, the execution time does not improve when using more cores.

Linux's `perf` performance counters indicate that there is a memory bottleneck; the number of executed instructions is for every run the same—even the number of created λ-terms by LambdaC++ is independent of the number of workers—but the number of cycles the cores stall on memory accesses increases. The figure also shows the speedup when artificially compensated for this effect, which is labeled 'w/o bottleneck'. In that case, we calculated the speedup when the instructions, which are measured during the x86 runs of LambdaC++, would have the same number of stall cycles as during the sequential version. The straight line suggests that the speedup trend of the first ten cores is continued, at least up to 24 cores. This shows that the applications scale properly to many cores, although with some constant overhead. This also suggests that the non-determinism in these experiments does not result in performance loss by doubly calculated terms, although we cannot measure it precisely without influencing the execution.

The speedup of `queens` shows a surprising trend: scaling to up to twelve cores give a superlinear speedup. Even more interesting is the fact that this holds for both the implementation in Haskell and LambdaC++. Since the 'w/o bottleneck' line is below the linear speedup, like for the other applications, we conclude that the memory hierarchy determines this unexpected measurement results. It might be the case that the amount of data or cache size plays a role, although we cannot find the exact cause.

The memory bottleneck is even more prominent on Starburst. The bandwidth

[2]If looked very carefully, the reader might notice that having two cores for LambdaC++ does not improve the performance. This is due to the structure of the program. In the implementation, the programs build up a list. Then, all but one worker concurrently compute the contents of this list, and one worker is dedicated to post-processing the list in-order, e.g., to generate output. In practice, post-processing takes less time than computation, so with two workers, one worker computes, and the other waits for its result.

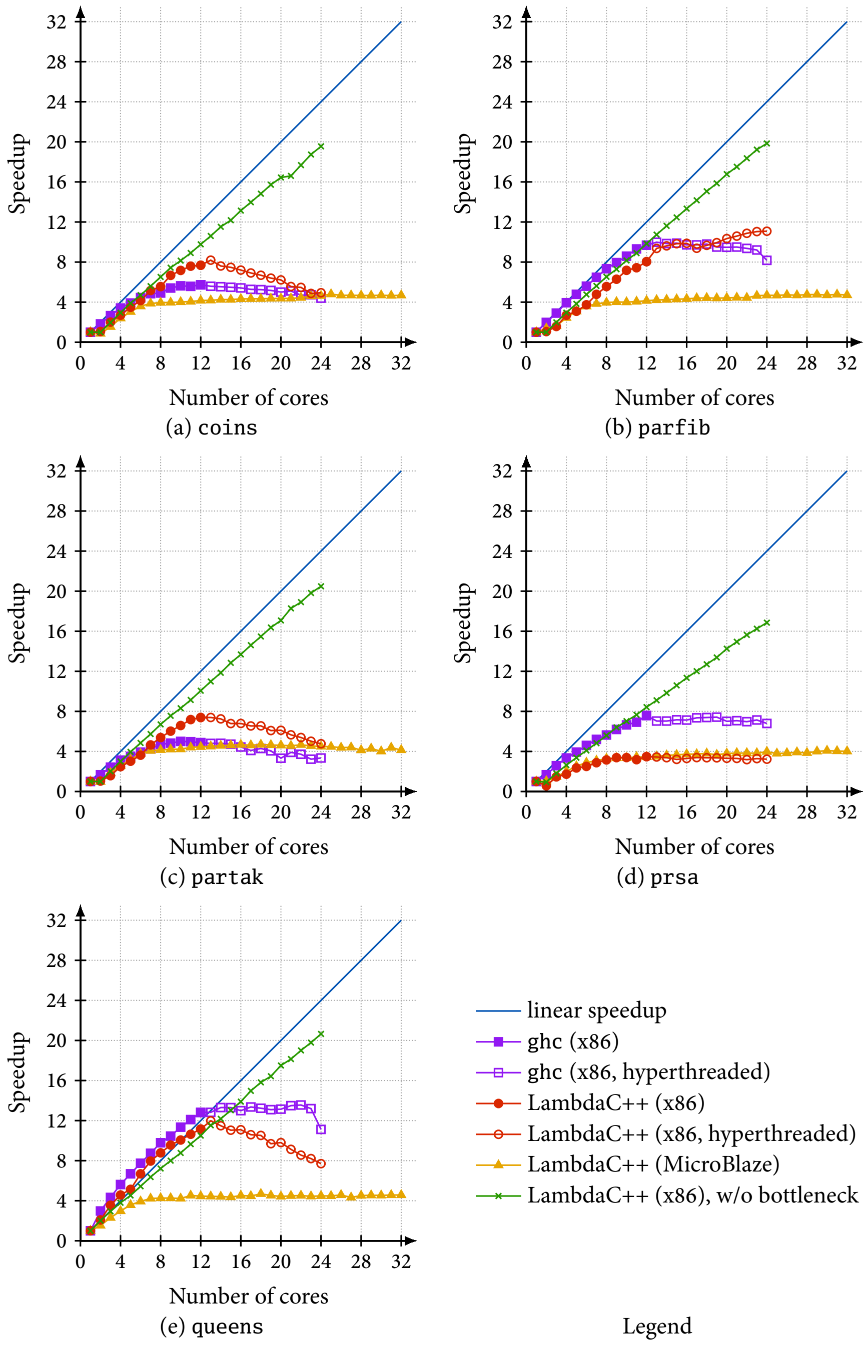


Figure 6.5 – Speedup of NoFib parallel benchmarks

Table 6.3 – Generated terms during evaluation (Lambda-C++, x86, 12 cores)

benchmark	local applications[a]	local constants[a]	globals[a]
`coins`	0.418	0.582	$1.36 \cdot 10^{-4}$
`parfib`	0.379	0.621	$1.44 \cdot 10^{-4}$
`partak`	0.351	0.648	$5.47 \cdot 10^{-4}$
`prsa`	0.412	0.583	$4.97 \cdot 10^{-3}$
`queens`	0.445	0.555	$9.10 \cdot 10^{-5}$

[a] Fraction of sum of all global and local terms

is saturated when eight cores are used. However, the same trend is visible; the workload scales properly to more cores, and the same amount of instructions is executed, but the cores just stall longer on every memory access. So, from a parallel-workload point of view, our proposed approach of avoiding usage of locks and allowing data races seems to be viable.

6.5.2 Locality and overhead

The memory bottleneck stems from the fact that all data, both global and local, are stored in main memory. Caches do keep data local, but eventually copy data to the main memory. In case of local terms, which are created, used, and destroyed by only one worker thread, this gives unnecessary memory traffic, and is therefore subject to future improvement. For every benchmark, we counted the amount of generated local function applications, local constants, and all global terms. The ratio of local and global data for LambdaC++ running on the Intel platform is listed in table 6.3. This table lists the measurements when using 12 cores, but the results are similar when another number of cores is used. The table shows that the number of local terms is orders of magnitude higher than that of the global terms.

If all local terms can be kept local, traffic to main memory and the effects of the memory bottleneck will be reduced significantly. Although untested, a solution could involve having a (large) scratchpad memory for every processor, and using this memory for all new local terms, i.e. the nursery of the GC. Anderson [7] reports that 99.8 % of the data does not survive that private nursery stage, so they are dead at the successive GC. Additionally, the optimum size for this memory is reported to range from 64 KB to 9 MB, depending on the application. The scratchpad memory can be backed by the main memory for longer-living terms. Such a modification to the RTS can be done transparently to the application. However, testing such a setup is left as future work.

Finally, the distribution of where time is spent during execution is measured. Figure 6.6 on the following page shows the most important states a worker can be in: global GC; local GC; stalling on a black hole, where another worker computes it;

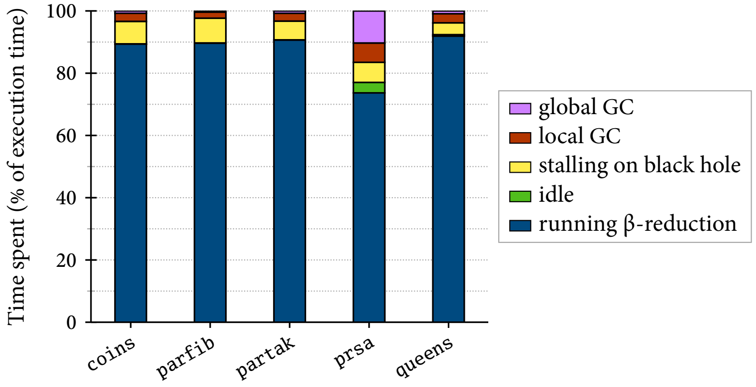


Figure 6.6 – Time spent during execution (LambdaC++, x86, 12 cores)

idle, because the work queue is empty; and running the application, which involves doing β-reductions, and represents the utilization of the application. The time is the sum of the time spent in such a phase, presented as a fraction of the combined total time of all workers. Only a small fraction is used for global GC, which is expected, because the number of global terms is much smaller than local ones. Interesting to see is that even local GC contributes only for 3.2 % of the total execution time.

6.6 Conclusion

One of the hardware design issues of a multiprocessor platform is atomic global communication between cores, such as cache coherency and synchronization. In this chapter, we showed that these hardware issues can be overcome at a different level. To this extent, we described a rather extreme example: a programming paradigm that allows an *atomic-free* implementation. Such an implementation does not rely on any read–modify–write operations or (mutex) locks, and does not rely on ordering guarantees of a strong memory model. We carefully introduced data races, even though the application keeps having a well-defined outcome.

For this, we implemented LambdaC++, a functional language that strictly follows the properties of λ-calculus. Since the language is single-assignment, synchronization is simplified. Expressions that can be evaluated concurrently, can safely be pushed onto and popped from a work queue, without proper synchronization. When work is lost due to a race condition during the push, it will eventually be calculated when required. Moreover, because the evaluation of an expression in λ-calculus always gives the same result, multiple workers might evaluate expressions concurrently, and the doubly calculated results are just garbage collected.

Based on the programming paradigm, we derived ordering rules to which the memory subsystem must adhere. These rules can be implemented by PMC, as defined in chapter 5. However, PMC assumes that data races should be avoided, where we introduce races in this chapter. Therefore, two additional memory operations (and realizations) are presented: non-exclusive write access, and a stronger fence. The former allows writing without having a lock on the object (and therefore allowing races), where the latter forces completion of earlier writes. The combination of these annotations and the behavior of λ-terms allows atomic-free execution on top of PMC. As a result, LambdaC++'s RTS can be used on top of any weak memory model that is supported by PMC, of which software cache coherency is exemplified in the experiments. This shows that *co-design* of the programming, concurrency, and memory model leads to solutions that allow less complex hardware, where modifications of only one of them is insufficient.

Applications written in LambdaC++ do not specify any annotation for the memory model; the memory model is completely removed from the programming model. Moreover, even concurrency is not the task of the programmer anymore. Although it is possible to give hints to the compiler what can be in done in parallel, and what might lead to higher performance when done sequentially, it is *impossible* to make errors regarding concurrency and synchronization. Therefore, the concurrency model is removed from the programming model too. The overview figure of this chapter on page 108 visualizes this in the overlap, which only covers the model of computation. Then, LambdaC++'s libraries are the glue logic that actually realize concurrency. This shows that the choice of the model of computation can implicitly allow concurrency in software, where the underlying layers are able to map it onto parallel hardware.

Although we have shown that an abstraction from concurrency can be made by using λ-calculus, we did not show that this leads to an *optimal* abstraction, in contrast to the memory model abstraction of chapter 5. Several aspects have to be considered. In contrast to fundamental minimal rules of memory operations and memory state changes, there is no such thing as a fundamental minimal or unified model for concurrency. Therefore, we can only show that a co-design approach can lead to interesting solutions, but we cannot say that this particular approach is the best. Experiments show that the performance of LambdaC++ is much lower than that of Haskell. A large part of this performance difference stems from the fact that we implemented C++ classes for a C++ compiler, which does not have knowledge of the functional properties of the application. Optimizations are mostly limited to machine instruction sequences and inlining C++ code, where GHC is able to analyze and optimize the functional program itself, before any machine instruction is emitted. We expect that any good concurrency abstraction model must allow analysis and optimization, such that the performance of the implementation of such a model can compete with hand-optimized code. A generalization of such a concurrency abstraction is left as future work.

7 Conclusion

Processors comprise an increasing number of cores. This trend constantly pushes hardware and software to their limits, and therefore forces both worlds to change. This thesis addresses several prominent issues related to hardware, software, and their interplay.

The hardware architecture drastically changes when the core count is increased. Where a few cores can share hardware resources relatively easily, many-core architectures face multiple issues that are related to an increased latency of communication. Latency between a core and the background memory is increased, because traversing the NoC takes multiple clock cycles. Furthermore, as shown in chapter 2, off-chip bandwidth to background memory cannot keep up with the growth of the memory bandwidth demands of all cores combined. As a result, shared SDRAM bandwidth becomes a scarce resource.

Most commercial many-core architectures include multiple levels of cache to keep as much code and data as possible close to the core that might need it. This setup offers a single shared-memory address space to software, but relies on coherency of the caches. It is complicated to realize coherency in hardware that requires multiple cycles to communicate a pending write, for example. From a software perspective, it is convenient when state changes happen instantly. However, concessions are made to the ease of use of such a memory architecture, as 'instantly' is hard to realize. Allowing a transient state, or even non-determinism in 'the' state of the memory, is easier to build, and therefore cheaper, and often allows a higher performance—a weak memory model is preferred over a strong one. A few architectures abandon hardware cache coherency altogether, or use scratchpad memories instead, and rely on software to communicate data between cores.

How to program such an architecture remains an open problem. Threading is a well-known concurrency model, and sounds like a straightforward way to exploit processing power of a parallel machine. Moreover, it fits to popular (imperative) programming languages. However, threading assumes communication via shared memory. As hardware tends to use weak memory models, this memory behaves in a non-straightforward way. Moreover, threading requires the programmer to split the computational task into chunks, and orchestrate the interaction manually. Manually programming threads on a few-core SPM processor might be viable, manually programming threads on a many-core architecture with a weak memory model is troublesome. Remarkably, all commercial platforms support (and advice) to program the platform using threads.

Hardware and software seems to be two separate worlds; hardware is often concerned with the behavior and performance of micro-architectural events or a single instruction, where software abstracts from the underlying hardware as much as possible. However, there is as much truth in the opposite of this separation. The behavior of a platform depends on how hardware and software interact, and how hardware abstractions match software models. In chapter 3, we discuss the relation of several abstraction layers: the memory model, the concurrency model, and the model of computation. A platform can be seen as the combination of the hardware and a specific set of abstraction layers on top, including the available compilers and run-time software that implement these layers. The programming model, of which the programming paradigm is a property of, can be seen as a peephole view on the platform. In turn, a programming language is an incarnation of the programming model, of which the specific form is determined by syntax, a type system, etc. The platform exposes details via the programming model to the programmer, which are essential to write application software. When many details are exposed, a programmer has full control over the behavior of the application, at the cost of work to fill in and control all these details, which can be error-prone. On the other hand, programming becomes easier when most details are hidden from the programmer, at the cost of possible overhead.

Chapter 4 presents optimizations in the many-core context, which are hidden from the programmer, namely the interconnect architecture and the implementation of synchronization. The interconnect is designed such that it is scalable in terms of hardware costs and performance. Synchronization, i.e. a mutex, is implemented such that it bypasses shared memory, and therefore relieves the memory bottleneck. From a programming point of view, the interface does not change, but there is a different trade-off how software should use the platform. For example, direct core-to-core (FIFO) communication is faster than communicating via shared memory, but data elements are limited in size. Additionally, locking an unlocked mutex is faster when it is done by the same process consecutively, than when multiple processes share the mutex—a higher locality of a mutex is preferred. Hence, these optimizations do not change the programming model, but might change the way how a programmer thinks about costs of operations, which in turn influences how a programmer uses the platform.

The techniques, as discussed in chapter 5, do change the programming model. A C programmer usually writes specific code for the target platform at hand, such as a fence to order store operations, and a cache flush to achieve coherency. This approach is not portable. For example, moving from an Intel to an ARM processor might require considerable modifications to the application. The chapter presents Portable Memory Consistency *(PMC)*, which abstracts from the memory model of the hardware. Then, the application's source code does not contain hardware-specific memory operations anymore. As a result, the programmer does not need to know the targeted memory model. The annotations required for PMC are directly related to the algorithm that is implemented. Therefore, the hardware's memory model is removed from the programming model, and PMC can be seen as a property of the abstract machine of the programming model. The translation from PMC to the actual hardware is done automatically. The key aspect of this abstraction is that it fits both the architecture trends of the underlying many-core hardware, and the requirements by the concurrency model on top.

Excluding the memory model of the hardware from the programming model, as discussed above, eliminates the chance to introduce bugs that are related to this memory model. Ideally, the concurrency model is excluded in the same way. Then, an application only defines the algorithm in terms defined by the model of computation, after which tooling can introduce concurrency automatically. The model of computation influences to what extent extracting concurrency is possible. In chapters 4 and 5, we used a programming model based on C. The abstract machine of C is based on the register machine model. However, a register machine is sequential in nature. Extracting concurrency from a sequential description is not straightforward. Therefore, it is easier to change the model of computation in order to hide the concurrency model.

In chapter 6, we use λ-calculus instead. This model does not define sequences of operations, but only dependencies and a rule that defines how to compute, which can be applied concurrently to the program. As a result, the compiler can analyze the program properly, after which the program can be executed using a concurrency model that suits the platform. In contrast, the (data) dependencies in a program written in C are often implicit and therefore unknown to the compiler, which makes transformations regarding concurrency compromise the functionality of the program. λ-calculus has the property that the computation of every expression is side-effect free; the outcome is always the same, regardless the state of the rest of the system. Therefore, this model of computation allows data races in distribution and communication of the computational load among cores. We show that it even allows atomic-free execution, where the program does not rely on any atomic sequence of reads and writes. This affects the underlying models: the memory subsystem does not have to guarantee a total order of specific state changes, and the hardware does not have to support atomic read–modify–write operations. Similar to PMC, a change in the model of computation influences other abstraction layers of the platform. Specifically, the requirements on the hardware are relaxed by means of atomic-free execution.

The central problem this thesis addresses, as formulated in section 1.4, is:

> *How can we cope with the hardware trends in embedded many-core architectures, from a programming perspective?*

The answer is that co-design of all abstraction layers of the platform and the programming model is required, such that the implementation as a whole matches the targeted properties of the system. The trends in hardware, as discussed in chapter 2, are reflected by these properties: shared memory is a scarce resource, data locality is important, a weak memory model used, and cache coherency (or scratchpad memory management) relies on software. To reduce the programming complexity, abstractions are needed that allow tooling to handle as much low-level properties as possible, such as software cache coherency and control over concurrency and synchronization. In terms of the platform model, as presented in chapter 3, the overlap of the programming model should be as small as possible to make programming many-core systems easier. However, there is no single 'perfect' platform or programming model, since it depends on the purpose of the system.

7.1 Contributions

The main contributions of this thesis are:

» *A coherent integration of several abstraction layers in a platform model and programming model*
A platform integrates the memory model, concurrency model, model of computation, and software in between to implement the conversion between all layers. This software includes the OS, run-time system, and compilers. The view a programmer has of the platform is captured by the programming model. This framework structures design choices and trade-offs of the implementation of all layers. (Chapter 3)

» *Warpfield, a scalable connectionless tree-shaped interconnect and ring*
The hardware costs of Warpfield scale linearly to the number of cores. This is a significant improvement over a connection-oriented network, which scales quadratic to the number of cores, as hardware resources are reserved for every pair of cores. Moreover, the performance of applications that communicate to shared memory via Warpfield, scales close to linear, as long as the memory bandwidth is not the bottleneck. Although Warpfield is connectionless and work-conserving, it has been shown that the packet latency through the tree-shaped network is bounded. Therefore, Warpfield is usable in a real-time context. (Chapter 4)

» *A distributed lock algorithm that exploits mutex locality and local memories*
Our lock algorithm bypasses shared memory, which is often a bottleneck in a many-core system. It uses message passing and the local (scratchpad) memories next to the core. This way, a mutex is realized, based on posted writes, and reads from a local memory only. (Chapter 4)

» *Portable Memory Consistency (PMC), a memory model abstraction*
PMC combines the requirements of threading and an imperative programming approach, and a many-core architecture with a weak memory model. It abstracts from the memory model of the hardware, and is therefore portable

between architectures with different memory models. PMC defines a memory model that is as weak as possible to allow maximum performance of the implementation, but strong enough to be able to simulate Sequential Consistency for data-race free applications. Moreover, the annotations of PMC allow a transparent implementation for common memory architectures, including software cache coherency and scratchpad memories. Therefore, PMC is the result of co-design of many-core hardware and the (threaded) concurrency model. (Chapter 5)

» *Scalable atomic-free implementation of λ-calculus*
We implemented LambdaC++, a functional language that closely follows the principles of λ-calculus. The rules that are derived from these principles allow atomic-free execution. Additionally, data races are introduced very carefully, which leads to non-determinism in the execution. However, this does not influence the outcome of the program. This shows that *co-design* based on the model of computation, concurrency model, and the memory model can lead to a property—in this case atomic-freedom—that could not have been realized by optimizations on any sole abstraction layer. However, this property is crucial in scalable many-core architectures. (Chapter 6)

» *Starburst, a many-core MicroBlaze system on FPGA*
Experiments are conducted on Starburst, which allows evaluation of the presented techniques in an environment that can be considered as harsh for C. It only supports a weak memory model, and it does not have hardware cache coherency and atomic read–modify–write instructions. In the current implementation, the shared memory bandwidth is a performance bottleneck. The project includes a flow that generates a SoC, given an architecture description that defines the type and number of cores and peripherals. Having this setup and actual applications running on an FPGA allows running long experiments at a speed that cannot be achieved in simulation. (Chapter 2)

7.2 Recommendations for future work

In chapter 6, we used a programming model based on λ-calculus to hide all lower layers from the programmer. The chapter showed that it is possible to do so, but did not address the performance loss due to the overhead of the abstraction. This is in contrast to PMC, which is designed such that the abstraction layer minimizes the amount of overhead it incurs. Future work should focus on this overhead. It is likely that the overhead is reduced considerably, when our compiler could optimize the program on a functional level. In practical solutions, streamlining programming by abstractions is only viable when these abstractions still allow performance optimizations or cost analysis, which can achieve an equivalent performance as hand-written or -optimized code.

At the moment, Starburst is a homogeneous system, based on general-purpose cores. In embedded systems, a system has usually a specific application domain.

For such a domain, it is often beneficial to add hardware accelerators to the SoC, such as an FFT component for DSP applications. Offloading tasks to accelerators, or even sharing accelerators by multiple processes, is now mostly a manual job. However, many complex questions regarding abstractions, compilation, and mapping remain open. For example, it is often good to map processes that intensively use an accelerator, physically close to that accelerator. Moreover, data streams to the accelerator and back to the (general-purpose) core can have a larger bandwidth than what core-to-core streams require, which could affect routing of these streams. It is unclear whether or how communication intensity and the related requirements and trade-offs can be determined automatically. Handling accelerators, or heterogeneity in general, is currently not defined by the platform model in this thesis. Future work might address accelerators from a programming perspective, such that the abstraction layer is able to hide mapping, routing, and sharing transparently, in a similar way as this thesis hides the hardware's memory model and the concurrency model.

Currently, the abstraction layers of the platform model only include functional aspects of the system. Non-functional aspects, such as timing, memory usage, and power usage, are not included. Since it is not possible to define non-functional constraints, compilers cannot take these non-functional aspects into account—a high performance is the usual optimization direction during compilation or execution. Future work might extend the platform model to include these aspects. It is quite probable that different non-functional aspects should be handled very differently. To address timing, for example, the maximum response time of the system might be included in the model of computation. The exact memory usage might be irrelevant, but the total memory usage should at least be lower than the total amount of memory in the system. On the other hand, (low) power usage could be just an optimization goal, but might also be subject to a strict budget. Integration of these aspects into the platform could give a better, or even automated, control over them.

In the suggestions above, it is essential to realize that optimizations at any level of a system influence other parts of the system. An efficient or properly balanced many-core architecture is unlikely to be achieved without co-design of multiple models, implementations of abstraction layers, and programming interfaces.

This empty page leaves some room for random thoughts:

Can a computer make errors? It only behaves according to the physical arrangement of matter, which is subject to the laws of nature. It is just that a 'broken' computer does not live up to the expectations of the user…

Etymology

A.1 Starburst

In astronomy, starburst *is a generic term to describe a region of space with an abnormally high rate of star formation. It is reserved for truly unusual objects.*[1]

Starburst is used as the name of the many-core architecture in this thesis, but the project also includes numerous scripts to automatically generate a complete SoC from a single architecture input file. Therefore, this project is more or less a starburst in embedded systems, because of the easy and quick method of creating new flavors of many-core SoCs, complete with network configuration, OS and bootstrap code.

At the moment, the project contains around 59 000 lines of C/C++ code, 14 000 lines of VHDL, 5000 lines of Makefiles, and 3000 lines of other sources, including assembly, Haskell and `gawk`. In comparison, this thesis consists of around 12 500 lines of LaTeX.

A real starburst is cluster NGC 3603, as depicted by figure A.1 on the following page[2]. The cover image of this thesis is the star-forming region LH 95 in the Large Magellanic Cloud[3]. Although this is an active region, it is not classified as a starburst.

A.2 Warpfield

The Alcubierre warp drive metric is a theoretic propulsion engine, which generates a field or 'bubble' of expansion and contraction of space around the spacecraft. Even though general relativity allows this type of propulsion, and it is a successful method in science-fiction series, any practical implementations is not expected

[1]Lemma 'Starburst region' of Wikipedia, October 2013

[2]`http://hubblesite.org/gallery/album/pr2007034b/`

[3]`http://hubblesite.org/gallery/album/pr2006055a/`

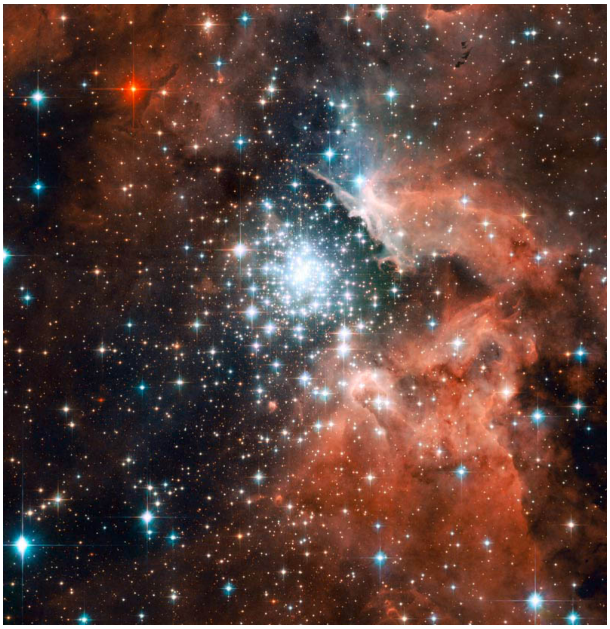


Figure A.1 – Starburst in cluster NGC 3603

soon[4]. As the Warpfield NoC targets high transmission speed, like most NoCs do, such a name seems appropriate for a subsystem of Starburst.

A.3 Helix

The operating system running on the MicroBlazes, called Helix, is named after the Helix Nebula (NGC 7293). Its appearance (see figure A.2[5]) explains its nick name: Eye of God. Mostly because of this name, it is appropriate to use it as the name of an OS.

A.4 `skat`

Programs running on Starburst, under supervision of Helix, have the name `skat` by default, just as `gcc` produces `a.out` by default. The name stems from the star Skat, or Delta Aquarii. This star is part of the constellation Aquarius—the same constellation the Helix Nebula is located in. Therefore, Helix watches over `skat`.

[4] H. White. *Warp Field Mechanics 101*, NASA, 2011.
[5] `http://hubblesite.org/gallery/album/pr2004032d/`

Figure A.2 – Helix Nebula

A.5 LambdaC++

If one reads the ++ as being an N, the name of the functional language in C++ coincides with Lambda Cen(tauri) Nebula. Figure A.3 on the following page[6] shows a picture of this star cluster, which is also designated IC 2944. Dark dust clouds are presumed to play a role in the bright nearby star formation. One can think of an analogy between this relation, and that of C++ and languages based on λ-calculus.

[6]`http://www.eso.org/public/images/eso1322a/`

Figure A.3 – IC 2944 cluster

This empty page leaves some room for random thoughts:

Concurrency cannot be invented by a programmer; it only makes his job harder.

Acronyms

*	$\prec$	set of operation orderings	*memory model*
	$\prec_F$	globally observable fence ordering	*memory model*
	$\prec_G$	globally observable ordering	*memory model*
	$\prec_L$	locally observable ordering	*memory model*
	$\prec_P$	globally observable program order	*memory model*
	$\prec_S$	globally observable synchronization ordering	*memory model*
	α	cache hit rate	
	β	buffer size after Warpfield's tree muxs (in flits)	*interconnect*
	τ	period of time	
	ψ	packet issue interval	*interconnect*
	E	execution	*memory model*
	H	number of flits that can hinder packets	*interconnect*
	I	issued packet count	*interconnect*
	O	set of operations	*memory model*
	P	set of processes	
	Q	set of packet request types	*interconnect*
	S_{bc}	best-case packet latency	*interconnect*
	S_{wc}	worst-case packet latency	*interconnect*
	V	set of locations	*memory model*
	W	last write	*memory model*
	ℓ	length of a packet (in flits)	*interconnect*
	n	number of cores in a SoC	
A	AC	Atomic Consistency	*memory model*
	ALU	arithmetic and logic unit	*hardware component*
	API	application programming interface	
	ASIC	application-specific integrated circuit	*hardware component*
B	BRAM	block RAM	*FPGA primitive*
C	CC	Cache Consistency	*memory model*
	CISC	complex instruction set computing	
	CMOS	complementary metal–oxide–semiconductor	
	CSDF	cyclo-static dataflow	

D	DC	Dag Consistency	*memory model*
	DDR	double data rate SDRAM	
	DMA	direct memory access	*hardware component*
	DRAM	dynamic RAM	
	DRF	data-race free	*software characteristic*
	DSM	distributed shared memory	*hardware architecture*
	DSP	digital signal processing	
	DVI	digital visual interface	
E	EC	Entry Consistency	*memory model*
F	FCFS	first-come-first-served	*arbitration*
	FF	flip-flop	*FPGA primitive*
	FIFO	first-in-first-out buffer	
	flit	flow control digit	*elementary NoC unit*
	FPGA	field-programmable gate array	*hardware component*
G	GC	garbage collection	
	GDO	global data order	*memory model property*
	GHC	Glasgow Haskell compiler	
	GP	general-purpose	
	GPO	global process order	*memory model property*
	GPU	graphics processing unit	*hardware component*
	GS	guaranteed-service	
	GS-LC	GS-Location Consistency	*memory model*
H	HW-CC	hardware cache coherency	
I	ID	identifier	
	ILP	instruction-level parallelism	*processor characteristic*
	IP	intellectual property	*hardware component*
K	KPN	Kahn process network	
L	LMB	local memory bus	*hardware component*
	LUT	lookup table	*FPGA primitive*
M	MLC	multi-level cache	*hardware architecture*
	MMU	memory management unit	*hardware component*
	mux	multiplexer	*hardware component*
N	NoC	network-on-chip	*hardware component*
	NUMA	non-uniform memory architecture	*hardware architecture*
O	OS	operating system	
P	PC	Processor Consistency	*memory model*
	PGAS	partitioned global address space	*hardware architecture*
	PLB	processor local bus	*hardware component*
	PMC	Portable Memory Consistency	*memory model*
	POSIX	portable operating system interface for Unix	
	PRAM	Pipelined RAM	*memory model*

R	RAM	random-access memory	
	RASP	random-access stored-program machine	
	RC	Release Consistency	*memory model*
	RISC	reduced instruction set computing	
	RMW	read–modify–write	
	RPC	remote procedure call	
	RTS	run-time system	
S	SANLP	static affine nested loop program	
	SC	Sequential Consistency	*memory model*
	SDF	synchronous dataflow	
	SDRAM	synchronous dynamic RAM	
	SIMD	single instruction, multiple data	*processor characteristic*
	SMP	symmetric multiprocessing	*hardware architecture*
	SoC	system-on-chip	*hardware architecture*
	SPM	scratchpad memory	*hardware component*
	SRAM	static RAM	
	SW-CC	software cache coherency	
T	TDM	time-division multiplexing	*arbitration*
	TSO	total store order	*memory model*
U	UART	universal asynchronous receiver/transmitter	
	UMA	uniform memory architecture	*hardware architecture*
	USB	universal serial bus	
V	VLIW	very large instruction word	*processor characteristic*
W	WC	Weak Consistency	*memory model*

Bibliography

[1] J. L. Abellán, J. Fernández, and M. E. Acacio. A G-line-based network for fast and efficient barrier synchronization in many-core CMPs. In *39th International Conference on Parallel Processing (ICPP 2010)*, pages 267–276. IEEE, Sept. 2010. doi: 10.1109/ICPP.2010.34. ISSN 0190-3918. (Cited on page 64).

[2] A. A. Abidi. The path to the software-defined radio receiver. *IEEE Journal of Solid-State Circuits*, 42(5):954–966, 2007. ISSN 0018-9200. doi: 10.1109/JSSC.2007.894307. (Cited on page 22).

[3] Adapteva. Epiphany-IV. URL `http://www.adapteva.com/products/silicon-devices/e64g401/`. (Cited on page 12).

[4] S. V. Adve and H.-J. Boehm. Memory models: a case for rethinking parallel languages and hardware. *Communications of the ACM*, 53(8):90–101, Aug. 2010. ISSN 0001-0782. doi: 10.1145/1787234.1787255. (Cited on pages 29, 31, 47, and 110).

[5] S. V. Adve, V. S. Adve, M. D. Hill, and M. K. Vernon. Comparison of hardware and software cache coherence schemes. In *Proceedings of the 18th annual international symposium on Computer architecture (ISCA '91)*, pages 298–308, New York, NY, USA, 1991. ACM. ISBN 0-89791-394-9. doi: 10.1145/115952.115982. (Cited on pages 17 and 101).

[6] M. Ahamad, R. A. Bazzi, R. John, P. Kohli, and G. Neiger. The power of processor consistency. In *Proceedings of the 5th annual ACM symposium on Parallel algorithms and architectures (SPAA '93)*, pages 251–260, New York, NY, USA, 1993. ACM. ISBN 0-89791-599-2. doi: 10.1145/165231.165264. (Cited on pages 80 and 93).

[7] T. A. Anderson. Optimizations in a private nursery-based garbage collector. In *Proceedings of the 2010 international symposium on Memory management (ISMM '10)*, pages 21–30, New York, NY, USA, 2010. ACM. doi: 10.1145/1806651.1806655. (Cited on pages 122 and 135).

[8] A. Andersson. Balanced search trees made simple. In F. Dehne, J.-R. Sack, N. Santoro, and S. Whitesides, editors, *Algorithms and Data Structures*, volume 709 of *Lecture Notes in Computer Science*, pages 60–71. Springer Berlin Heidelberg, 1993. ISBN 978-3-540-57155-1. doi: 10.1007/3-540-57155-8_236. (Cited on page 68).

[9] Apple. Introducing blocks and Grand Central Dispatch, 2010. URL `http://developer.apple.com/library/mac/#featuredarticles/BlocksGCD/`. (Cited on page 32).

[10] J. Armstrong, R. Virding, C. Wikström, and M. Williams. Concurrent programming in ERLANG. 1993. ISBN 0-13-508301-X. (Cited on page 112).

[11] M. Batty, S. Owens, S. Sarkar, P. Sewell, and T. Weber. Mathematizing C++ concurrency. In *Proceedings of the 38th annual ACM SIGPLAN-SIGACT symposium on Principles of programming languages (POPL '11)*, pages 55–66. ACM, 2011. ISBN 978-1-4503-0490-0. doi: 10.1145/1926385.1926394. (Cited on pages 36 and 85).

[12] B. N. Bershad, M. J. Zekauskas, and W. A. Sawdon. The Midway distributed shared memory system. In *Compcon Spring '93, Digest of Papers*, pages 528–537, Feb. 1993. doi: 10.1109/CMPCON.1993.289730. (Cited on page 83).

[13] A. Bhattacharjee, G. Contreras, and M. Martonosi. Parallelization libraries: Characterizing and reducing overheads. *ACM Transactions on Architecture and Code Optimization (TACO)*, 8(1):5:1–5:29, Feb. 2011. ISSN 1544-3566. doi: 10.1145/1952998.1953003. (Cited on page 112).

[14] C. Bienia. *Benchmarking Modern Multiprocessors*. PhD thesis, Princeton University, Princeton, NJ, USA, Jan. 2011. ISBN 978-1-124-49186-8. (Cited on page 24).

[15] T. Bijlsma, M. J. G. Bekooij, and G. J. M. Smit. Circular buffers with multiple overlapping windows for cyclic task graphs. In *Transactions on High-Performance Embedded Architectures (HiPEAC)*, volume 5, issue 3. 2011. URL `http://www.hipeac.net/system/files?file=paper71.pdf`. Special issue on SAMOS 2009 International Symposium on Systems, Architectures, Modeling and Simulation. (Cited on page 102).

[16] T. Bjerregaard and S. Mahadevan. A survey of research and practices of network-on-chip. *ACM Computing Surveys (CSUR)*, 38(1), June 2006. ISSN 0360-0300. doi: 10.1145/1132952.1132953. (Cited on pages 45 and 46).

[17] G. Blake, R. G. Dreslinski, and T. Mudge. A survey of multicore processors. *Signal Processing Magazine, IEEE*, 26(6):26–37, Nov. 2009. ISSN 1053-5888. doi: 10.1109/MSP.2009.934110. (Cited on pages 14 and 17).

[18] R. D. Blumofe, C. F. Joerg, B. C. Kuszmaul, C. E. Leiserson, K. H. Randall, and Y. Zhou. Cilk: an efficient multithreaded runtime system. In *Proceedings of the fifth ACM SIGPLAN symposium on Principles and practice of parallel programming (PPOPP '95)*, pages 207–216, New York, NY, USA, 1995. ACM. ISBN 0-89791-700-6. doi: 10.1145/209936.209958. (Cited on page 41).

[19] R. D. Blumofe, M. Frigo, C. F. Joerg, C. E. Leiserson, and K. H. Randall. DAG-consistent distributed shared memory. In *Proceedings of the 10th International Parallel Processing Symposium (IPPS '96)*, pages 132–141, Apr. 1996. doi: 10.1109/IPPS.1996.508049. (Cited on page 95).

[20] H.-J. Boehm. Threads cannot be implemented as a library. In *Proceedings of the 2005 ACM SIGPLAN conference on Programming language design and implementation (PLDI '05)*, PLDI '05, pages 261–268, New York, NY, USA, 2005. ACM. ISBN 1-59593-056-6. doi: 10.1145/1065010.1065042. (Cited on pages 3, 36, and 41).

[21] A. Boeijink, P. K. F. Hölzenspies, and J. Kuper. Introducing the PilGRIM: A processor for executing lazy functional languages. In J. Hage and M. T. Morazán, editors, *Implementation and Application of Functional Languages*, volume 6647 of *Lecture Notes in Computer Science*, pages 54–71. Springer Berlin Heidelberg, 2011. ISBN 978-3-642-24275-5. doi: 10.1007/978-3-642-24276-2_4. (Cited on page 112).

[22] S. Borkar and A. A. Chien. The future of microprocessors. *Communications of the ACM*, 54(5):67–77, May 2011. ISSN 0001-0782. doi: 10.1145/1941487.1941507. (Cited on pages 2, 12, 13, 17, and 23).

[23] C. Brooks, E. A. Lee, X. Liu, S. Neuendorffer, Y. Zhao, and H. Zheng. Heterogeneous concurrent modeling and design in Java (volume 1: Introduction to Ptolemy II). Technical Report UCB/EECS-2007-7, EECS Department, University of California, Berkeley, Jan. 2007. URL `http://www.eecs.berkeley.edu/Pubs/TechRpts/2007/EECS-2007-7.html`. (Cited on page 33).

[24] C++11. ISO/IEC 14882:2011. (Cited on pages 36 and 85).

[25] C99. ISO/IEC 9899:1999. (Cited on page 36).

[26] M. R. Casu, M. R. Roch, S. V. Tota, and M. Zamboni. A NoC-based hybrid message-passing/shared-memory approach to CMP design. *Microprocessors and Microsystems*, 35(2):261–273, Mar. 2011. ISSN 0141-9331. doi: 10.1016/j.micpro.2010.09.006. (Cited on page 64).

[27] Cavium. OCTEON II CN68XX. URL `http://www.cavium.com/OCTEON-II_CN68XX.html`. (Cited on page 12).

[28] M. M. T. Chakravarty, R. Leshchinskiy, S. Peyton Jones, G. Keller, and S. Marlow. Data parallel Haskell: a status report. In *Proceedings of the 2007 workshop on Declarative aspects of multicore programming (DAMP '07)*, pages 10–18, New York, NY, USA, 2007. ACM. ISBN 978-1-59593-690-5. doi: 10.1145/1248648.1248652. (Cited on page 112).

[29] B. Choi, R. Komuravelli, H. Sung, R. Smolinski, N. Honarmand, S. V. Adve, V. S. Adve, N. P. Carter, and C.-T. Chou. DeNovo: Rethinking the memory hierarchy for disciplined parallelism. In *International Conference on Parallel Architectures and Compilation Techniques (PACT 2011)*, pages 155–166, Oct. 2011. doi: 10.1109/PACT.2011.21. (Cited on pages 3, 4, 17, and 23).

[30] A. Church. An unsolvable problem of elementary number theory. *American journal of mathematics*, 58(2):345–363, 1936. (Cited on page 34).

[31] A. Church and J. B. Rosser. Some properties of conversion. *Transactions of the American Mathematical Society*, 39(3):472–482, 1936. ISSN 00029947. URL `http://www.jstor.org/stable/1989762`. (Cited on page 114).

[32] Clojure, 2007. URL `http://clojure.org`. (Cited on page 112).

[33] S. A. Cook and R. A. Reckhow. Time bounded random access machines. *Journal of Computer and System Sciences*, 7(4):354–375, Aug. 1973. ISSN 0022-0000. doi: 10.1016/S0022-0000(73)80029-7. (Cited on page 33).

[34] CoreMark. URL `http://www.coremark.org`. (Cited on pages 13 and 14).

[35] D. Culler, J. P. Singh, and A. Gupta. *Parallel Computer Architecture: A Hardware/Software Approach*. Morgan Kaufmann Publishers, Inc., Sept. 1998. ISBN 1-55860-343-3. (Cited on pages 15, 17, 23, 29, 60, and 112).

[36] L. Dagum and R. Menon. OpenMP: an industry standard API for shared-memory programming. *IEEE Computational Science Engineering*, 5(1):46–55, Jan.-Mar. 1998. ISSN 1070-9924. doi: 10.1109/99.660313. (Cited on pages 3 and 17).

[37] G. De Micheli, C. Seiculescu, S. Murali, et al. Networks on chips: from research to products. In *Proceedings of the 47th Design Automation Conference (DAC 2010)*, pages 300–305, New York, NY, USA, 2010. ACM. doi: 10.1145/1837274.1837352. (Cited on page 46).

[38] J. Dean and S. Ghemawat. Mapreduce: simplified data processing on large clusters. *Communications of the ACM*, 51(1):107–113, Jan. 2008. ISSN 0001-0782. doi: 10.1145/1327452.1327492. (Cited on pages 32 and 113).

[39] B. H. J. Dekens, P. Wilmanns, M. J. G. Bekooij, and G. J. M. Smit. Low-cost guaranteed-throughput communication ring for real-time streaming MPSoCs. In *Conference on Design and Architectures for Signal and Image Processing (DASIP)*. IEEE, Oct. 2013. ISBN 979-10-92279-01-6. (Cited on page 53).

[40] K. Denolf, M. J. G. Bekooij, J. Cockx, D. Verkest, and H. Corporaal. Exploiting the expressiveness of cyclo-static dataflow to model multimedia implementations. *EURASIP Journal on Advances in Signal Processing*, June 2007. doi: 10.1155/2007/84078. (Cited on page 102).

[41] M. Dubois, C. Scheurich, and F. Briggs. Memory access buffering in multiprocessors. *SIGARCH Computer Architecture News*, 14(2):434–442, May 1986. ISSN 0163-5964. doi: 10.1145/17356.17406. (Cited on page 82).

[42] Freescale. T4240, 2012. URL `http://www.freescale.com/webapp/sps/site/prod_summary.jsp?code=T4240`. (Cited on page 12).

[43] M. Frigo. The weakest reasonable memory model. Master's thesis, MIT Department of EE and CS, Jan. 1998. (Cited on pages 92 and 93).

[44] G. R. Gao and V. Sarkar. Location consistency—a new memory model and cache consistency protocol. *IEEE Transactions on Computers*, 49(8):798–813, Aug. 2000. ISSN 0018-9340. doi: 10.1109/12.868026. (Cited on pages 84 and 93).

[45] K. Gharachorloo, D. Lenoski, J. Laudon, P. Gibbons, A. Gupta, and J. Hennessy. Memory consistency and event ordering in scalable shared-memory multiprocessors. *SIGARCH Computer Architecture News*, 18(3a):15–26, May 1990. ISSN 0163-5964. doi: 10.1145/325096.325102. (Cited on pages 83, 85, 93, and 106).

[46] GHC. The Glasgow Haskell Compiler. URL `http://www.haskell.org/ghc/`. (Cited on page 34).

[47] K. Goossens and A. Hansson. The aethereal network on chip after ten years: Goals, evolution, lessons, and future. In *Proceedings of the 47th Design Automation Conference (DAC '10)*, pages 306–311, New York, NY, USA, June 2010. ACM. doi: 10.1145/1837274.1837353. (Cited on pages 15, 16, 46, 48, and 50).

[48] C. Grelck. Shared memory multiprocessor support for SAC. In K. Hammond, T. Davie, and C. Clack, editors, *Implementation of Functional Languages*, volume 1595 of *Lecture Notes in Computer Science*, pages 38–53. Springer Berlin Heidelberg, 1999. ISBN 978-3-540-66229-7. doi: 10.1007/3-540-48515-5_3. (Cited on page 112).

[49] P. Guerrier and A. Greiner. A generic architecture for on-chip packet-switched interconnections. In *Proceedings of the conference on Design, Automation and Test in Europe (DATE '00)*, pages 250–256, New York, NY, USA, 2000. ACM. ISBN 1-58113-244-1. doi: 10.1145/343647.343776. (Cited on page 15).

[50] K. Hammond and G. Michaelson. Hume: A domain-specific language for real-time embedded systems. In F. Pfenning and Y. Smaragdakis, editors, *Generative Programming and Component Engineering*, volume 2830 of *Lecture Notes in Computer Science*, pages 37–56. Springer Berlin Heidelberg, Sept. 2003. ISBN 978-3-540-20102-1. doi: 10.1007/978-3-540-39815-8_3. (Cited on page 42).

[51] A. Hansson, K. Goossens, M. Bekooij, and J. Huisken. CoMPSoC: A template for composable and predictable multi-processor system on chips. *ACM Transactions on Design Automation of Electronic Systems (TODAES)*, 14(1):2:1–2:24, Jan. 2009. ISSN 1084-4309. doi: 10.1145/1455229.1455231. (Cited on pages 15 and 41).

[52] F. Henderson. Accurate garbage collection in an uncooperative environment. In *Proceedings of the 3rd international symposium on Memory management (ISMM '02)*, pages 150–156, New York, NY, USA, 2002. ACM. ISBN 1-58113-539-4. doi: 10.1145/512429.512449. (Cited on page 123).

[53] M. Herlihy. Wait-free synchronization. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 13(1):124–149, Jan. 1991. ISSN 0164-0925. doi: 10.1145/114005.102808. (Cited on page 60).

[54] M. Herlihy. A methodology for implementing highly concurrent data objects. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 15(5):745–770, Nov. 1993. ISSN 0164-0925. doi: 10.1145/161468.161469. (Cited on page 125).

[55] M. D. Hill. Multiprocessors should support simple memory consistency models. *Computer*, 31(8):28–34, Aug. 1998. ISSN 0018-9162. doi: 10.1109/2.707614. (Cited on pages 30 and 85).

[56] G. J. Holzmann. The model checker SPIN. *IEEE Transactions on Software Engineering*, 23(5):279–295, Aug. 1997. ISSN 0098-5589. doi: 10.1109/32.588521. (Cited on page 68).

[57] House, 2006. URL `http://programatica.cs.pdx.edu/House/`. (Cited on page 112).

[58] J. Howard, S. Dighe, Y. Hoskote, S. Vangal, D. Finan, G. Ruhl, D. Jenkins, H. Wilson, N. Borkar, G. Schrom, F. Pailet, S. Jain, T. Jacob, S. Yada, S. Marella, P. Salihundam, V. Erraguntla, M. Konow, M. Riepen, G. Droege, J. Lindemann, M. Gries, T. Apel, K. Henriss, T. Lund-Larsen, S. Steibl, S. Borkar, V. De, R. Van der Wijngaart, and T. Mattson. A 48-core IA-32 message-passing processor with DVFS in 45nm CMOS. In *IEEE International Solid-State Circuits Conference Digest of Technical Papers (ISSCC 2010)*, pages 108–109, Feb. 2010. doi: 10.1109/ISSCC.2010.5434077. (Cited on pages 12 and 64).

[59] P. W. Hutto and M. Ahamad. Slow memory: weakening consistency to enhance concurrency in distributed shared memories. In *Proceedings of the 10th International Conference on Distributed Computing Systems (ICDCS 1990)*, pages 302–309, May 1990. doi: 10.1109/ICDCS.1990.89297. (Cited on page 82).

[60] Intel. i7-3930K, 2011. URL `http://www.intel.com/content/www/us/en/processor-comparison/processor-specifications.html?proc=63697`. (Cited on page 12).

[61] Intel. Xeon Phi, 2012. URL `http://software.intel.com/en-us/mic-developer`. (Cited on page 12).

[62] A. A. Jerraya, A. Bouchhima, and F. Pétrot. Programming models and HW-SW interfaces abstraction for multi-processor SoC. In *Proceedings of the 43rd annual Design Automation Conference (DAC '06)*, pages 280–285, New York, NY, USA, 2006. ACM. ISBN 1-59593-381-6. doi: 10.1145/1146909.1146981. (Cited on page 42).

[63] T. Johnsson. Lambda lifting: Transforming programs to recursive equations. In J.-P. Jouannaud, editor, *Functional Programming Languages and Computer Architecture*, volume 201 of *Lecture Notes in Computer Science*, pages 190–203. Springer Berlin Heidelberg, Sept. 1985. ISBN 978-3-540-15975-9. doi: 10.1007/3-540-15975-4_37. (Cited on page 119).

[64] R. Jones, A. Hosking, and E. Moss. *The Garbage Collection Handbook: The Art of Automatic Memory Management*. Chapman and Hall, Jan. 2012. ISBN 978-1-4200-8279-1. (Cited on page 123).

[65] G. Kahn. The semantics of a simple language for parallel programming. In *Information Processing '74*, pages 471–475. North-Holland Publishing Company, 1974. (Cited on page 31).

[66] R. Kalla, B. Sinharoy, W. J. Starke, and M. Floyd. Power7: IBM's next-generation server processor. *IEEE Micro*, 30(2):7–15, Apr. 2010. ISSN 0272-1732. doi: 10.1109/MM.2010.38. (Cited on page 12).

[67] D. Kranz, K. Johnson, A. Agarwal, et al. Integrating message-passing and shared-memory: early experience. *ACM SIGPLAN Notices*, 28(7):54–63, July 1993. doi: 10.1145/173284.155338. (Cited on page 17).

[68] S. Kwon, Y. Kim, W.-C. Jeun, S. Ha, and Y. Paek. A retargetable parallel-programming framework for MPSoC. *ACM Transactions on Design Automation of Electronic Systems (TODAES)*, 13(3):39:1–39:18, July 2008. ISSN 1084-4309. doi: 10.1145/1367045.1367048. (Cited on page 41).

[69] L. Lamport. A new solution of Dijkstra's concurrent programming problem. *Communications of the ACM*, 17(8):453–455, Aug. 1974. ISSN 0001-0782. doi: 10.1145/361082.361093. (Cited on pages 61 and 124).

[70] L. Lamport. How to make a multiprocessor computer that correctly executes multiprocess programs. *IEEE Transactions on Computers*, C-28(9):690–691, Sept. 1979. ISSN 0018-9340. doi: 10.1109/TC.1979.1675439. (Cited on pages 30 and 80).

[71] E. A. Lee. The problem with threads. *Computer*, 39(5):33–42, May 2006. ISSN 0018-9162. doi: 10.1109/MC.2006.180. (Cited on pages 3, 38, 41, and 110).

[72] E. A. Lee and T. M. Parks. Dataflow process networks. *Proceedings of the IEEE*, 83 (5):773–801, May 1995. ISSN 0018-9219. doi: 10.1109/5.381846. (Cited on page 32).

[73] M. D. Linderman, J. D. Collins, H. Wang, and T. H. Meng. Merge: a programming model for heterogeneous multi-core systems. *SIGPLAN Notices*, 43(3):287–296, Mar. 2008. ISSN 0362-1340. doi: 10.1145/1353536.1346318. (Cited on page 41).

[74] R. J. Lipton and J. S. Sandberg. PRAM: A scalable shared memory. Technical Report CS-TR-180-88, Princeton University, Sept. 1988. URL `http://www.cs.princeton.edu/research/techreps/TR-180-88`. (Cited on page 82).

[75] G. Long, N. Yuan, and D. Fan. Location Consistency model revisited: Problem, solution and prospects. In *9th International Conference on Parallel and Distributed Computing, Applications and Technologies (PDCAT 2008)*, pages 91–98, Dec. 2008. doi: 10.1109/PDCAT.2008.31. (Cited on page 84).

[76] R. Loogen, Y. Ortega-Mallén, and R. Peña-Marí. Parallel functional programming in Eden. *Journal of Functional Programming*, 15(3):431–475, May 2005. ISSN 1469-7653. doi: 10.1017/S0956796805005526. (Cited on page 112).

[77] S. Marlow, S. Peyton Jones, and S. Singh. Runtime support for multicore Haskell. *ACM SIGPLAN Notices*, 44(9):65–78, Aug. 2009. ISSN 0362-1340. doi: 10.1145/1631687.1596563. (Cited on page 112).

[78] B. W. Meister, P. Janson, and L. Svobodova. Connection-oriented versus connectionless protocols: A performance study. *IEEE Transactions on Computers*, C-34 (12):1164–1173, Dec. 1985. ISSN 0018-9340. doi: 10.1109/TC.1985.6312214. (Cited on page 45).

[79] J. M. Mellor-Crummey and M. L. Scott. Algorithms for scalable synchronization on shared-memory multiprocessors. *ACM Transactions on Computer Systems (TOCS)*, 9(1):21–65, Feb. 1991. ISSN 0734-2071. doi: 10.1145/103727.103729. (Cited on page 60).

[80] R. Milner, J. Parrow, and D. 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. (Cited on page 34).

[81] D. Molka, D. Hackenberg, R. Schone, and M. S. Muller. Memory performance and cache coherency effects on an Intel Nehalem multiprocessor system. In *18th International Conference on Parallel Architectures and Compilation Techniques (PACT '09)*, pages 261–270, 2009. doi: 10.1109/PACT.2009.22. (Cited on pages 3 and 4).

[82] M. Monchiero, G. Palermo, C. Silvano, and O. Villa. Power/performance hardware optimization for synchronization intensive applications in MPSoCs. In *Proceedings of Design, Automation and Test in Europe (DATE '06)*, pages 606–611. European Design and Automation Association, 2006. doi: 10.1109/DATE.2006.243994. (Cited on page 64).

[83] D. Mosberger. Memory consistency models. *ACM SIGOPS Operating Systems Review*, 27(1):18–26, Jan. 1993. doi: 10.1145/160551.160553. (Cited on pages 30 and 80).

[84] R. Nasre, M. Burtscher, and K. Pingali. Atomic-free irregular computations on GPUs. In *Proceedings of the 6th Workshop on General Purpose Processor Using Graphics Processing Units (GPGPU-6)*, pages 96–107, New York, NY, USA, 2013. ACM. ISBN 978-1-4503-2017-7. doi: 10.1145/2458523.2458533. (Cited on page 113).

[85] N. Nguyen, A. Dominguez, and R. Barua. Memory allocation for embedded systems with a compile-time-unknown scratch-pad size. *ACM Transactions on Embedded Computing Systems (TECS)*, 8(3):21:1–21:32, Apr. 2009. ISSN 1539-9087. doi: 10.1145/1509288.1509293. (Cited on page 14).

[86] A. Nieuwland, J. Kang, O. P. Gangwal, R. Sethuraman, N. Busá, K. Goossens, R. Peset Llopis, and P. Lippens. C-HEAP: A heterogeneous multi-processor architecture template and scalable and flexible protocol for the design of embedded signal processing systems. *Design Automation for Embedded Systems*, 7(3):233–270, Oct. 2002. ISSN 0929-5585. doi: 10.1023/A:1019782306621. (Cited on page 22).

[87] H. Nikolov, T. Stefanov, and E. Deprettere. Systematic and automated multiprocessor system design, programming, and implementation. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems (TCAD)*, 27(3):542–555, Mar. 2008. ISSN 0278-0070. doi: 10.1109/TCAD.2007.911337. (Cited on pages 38 and 41).

[88] NoFib. Haskell benchmark suite, 2010. URL `http://darcs.haskell.org/nofib/`. (Cited on pages 25 and 132).

[89] F. Ophelders, M. J. G. Bekooij, and H. Corporaal. A tuneable software cache coherence protocol for heterogeneous MPSoCs. In *Proceedings of the 7th IEEE/ACM international conference on Hardware/software codesign and system synthesis (CODES+ISSS '09)*, pages 383–392, New York, NY, USA, 2009. ACM. doi: 10.1145/1629435.1629488. (Cited on page 61).

[90] M. Pelcat, S. Aridhi, J. Piat, and J.-F. Nezan. Dataflow model of computation. In *Physical Layer Multi-Core Prototyping*, volume 171 of *Lecture Notes in Electrical Engineering*, chapter 3, pages 53–75. Springer London, 2013. ISBN 978-1-4471-4209-6. doi: 10.1007/978-1-4471-4210-2_3. (Cited on page 23).

[91] F. Pétrot, A. Greiner, and P. Gomez. On cache coherency and memory consistency issues in NoC based shared memory multiprocessor SoC architectures. In *9th EUROMICRO Conference on Digital System Design: Architectures, Methods and Tools (DSD 2006)*, pages 53–60, 2006. doi: 10.1109/DSD.2006.73. (Cited on page 17).

[92] S. Peyton Jones, A. Gordon, and S. Finne. Concurrent Haskell. In *Proceedings of the 23rd ACM SIGPLAN-SIGACT symposium on Principles of programming languages (POPL '96)*, pages 295–308, New York, NY, USA, 1996. ACM. doi: 10.1145/237721.237794. (Cited on page 112).

[93] R. Pike. The Go programming language, Oct. 2009. URL `http://golang.org`. (Cited on page 36).

[94] Pthread. The Open Group Base Specifications Issue 6, IEEE Std 1003.1, 2004 Edition. URL `http://pubs.opengroup.org/onlinepubs/007904975/`. Section 4.10 and A.4.10. (Cited on pages 18 and 31).

[95] L. Sha, R. Rajkumar, and J. P. Lehoczky. Priority inheritance protocols: an approach to real-time synchronization. *IEEE Transactions on Computers*, 39(9):1175–1185, Sept. 1990. ISSN 0018-9340. doi: 10.1109/12.57058. (Cited on page 20).

[96] J. L. Shin, D. Huang, B. Petrick, C. Hwang, K. W. Tam, A. Smith, H. Pham, H. Li, T. Johnson, F. Schumacher, A. S. Leon, and A. Strong. A 40 nm 16-core 128-thread SPARC SoC processor. *IEEE Journal of Solid-State Circuits*, 46(1):131–144, Oct. 2011. ISSN 0018-9200. doi: 10.1109/JSSC.2010.2080491. (Cited on page 12).

[97] M. Singhal. A taxonomy of distributed mutual exclusion. *Journal of Parallel and Distributed Computing*, 18(1):94–101, May 1993. ISSN 0743-7315. doi: 10.1006/jpdc.1993.1048. (Cited on page 64).

[98] K. C. Sivaramakrishnan, L. Ziarek, R. Prasad, and S. Jagannathan. Lightweight asynchrony using parasitic threads. In *Proceedings of the 5th ACM SIGPLAN workshop on Declarative aspects of multicore programming (DAMP '10)*, pages 63–72, New York, NY, USA, 2010. ACM. ISBN 978-1-60558-859-9. doi: 10.1145/1708046.1708059. (Cited on page 112).

[99] D. B. Skillicorn and D. Talia. Models and languages for parallel computation. *ACM Computing Surveys*, 30(2):123–169, June 1998. ISSN 0360-0300. doi: 10.1145/280277.280278. (Cited on page 33).

[100] M. Steine, M. Bekooij, and M. Wiggers. A priority-based budget scheduler with conservative dataflow model. In *12th Euromicro Conference on Digital System Design, Architectures, Methods and Tools (DSD '09)*, pages 37–44, Aug. 2009. doi: 10.1109/DSD.2009.148. (Cited on page 20).

[101] R. C. Steinke and G. J. Nutt. A unified theory of shared memory consistency. *Journal of the ACM (JACM)*, 51(5):800–849, Sept. 2004. ISSN 0004-5411. doi: 10.1145/1017460.1017464. (Cited on pages 30, 82, 85, 86, and 92).

[102] P. Stenström. A survey of cache coherence schemes for multiprocessors. *Computer*, 23(6):12–24, June 1990. ISSN 0018-9162. doi: 10.1109/2.55497. (Cited on page 30).

[103] C. Stoif, M. Schoeberl, B. Liccardi, and J. Haase. Hardware synchronization for embedded multi-core processors. In *IEEE International Symposium on Circuits and Systems (ISCAS)*, pages 2557–2560, 2011. doi: 10.1109/ISCAS.2011.5938126. (Cited on page 64).

[104] V. Suhendra, A. Roychoudhury, and T. Mitra. Scratchpad allocation for concurrent embedded software. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 32(4):13:1–13:47, Apr. 2010. ISSN 0164-0925. doi: 10.1145/1734206.1734210. (Cited on page 14).

[105] H. Sutter and J. Larus. Software and the concurrency revolution. *Queue*, 3(7):54–62, Sept. 2005. ISSN 1542-7730. doi: 10.1145/1095408.1095421. (Cited on page 3).

[106] W. Thies, M. Karczmarek, and S. P. Amarasinghe. StreamIt: A language for streaming applications. In *Compiler Construction*, volume 2304 of *Lecture Notes in Computer Science*, pages 179–196. Springer Berlin Heidelberg, 2002. doi: 10.1007/3-540-45937-5_14. (Cited on page 23).

[107] Tilera. TILE-Gx. URL `http://www.tilera.com/products/processors/TILE-Gx_Family`. (Cited on page 12).

[108] J. J. Tithi, D. Matani, G. Menghani, and R. A. Chowdhury. Avoiding locks and atomic instructions in shared-memory parallel BFS using optimistic parallelization. In *Proceedings of the Workshop on Multithreaded Architectures and Applications (MTAAP 2013)*, pages 1628–1637. IEEE Computer Society, May 2013. ISBN 978-0-7695-4979-8. doi: 10.1109/IPDPSW.2013.241. (Cited on page 113).

[109] A. Tumeo, C. Pilato, G. Palermo, F. Ferrandi, and D. Sciuto. HW/SW methodologies for synchronization in fpga multiprocessors. In *Proceedings of the ACM/SIGDA international symposium on Field programmable gate arrays (FPGA '09)*, pages 265–268, New York, NY, USA, 2009. ACM. ISBN 978-1-60558-410-2. doi: 10.1145/1508128.1508174. (Cited on page 64).

[110] J. W. van den Brand and M. Bekooij. Streaming consistency: a model for efficient MPSoC design. In *10th Euromicro Conference on Digital System Design Architectures, Methods and Tools (DSD 2007)*, pages 27–34, Aug. 2007. doi: 10.1109/DSD.2007.4341446. (Cited on page 83).

[111] P. van der Wolf, E. de Kock, T. Henriksson, et al. Design and programming of embedded multiprocessors: an interface-centric approach. In *Proceedings of the 2nd IEEE/ACM/IFIP international conference on Hardware/software codesign and system synthesis (CODES+ISSS '04)*, pages 206–217, New York, NY, USA, 2004. ACM. ISBN 1-58113- 937-3. doi: 10.1145/1016720.1016771. (Cited on page 17).

[112] S. Verdoolaege, H. Nikolov, and T. Stefanov. pn: a tool for improved derivation of process networks. *EURASIP Journal on Embedded Systems*, 2007, Jan. 2007. ISSN 1687-3955. doi: 10.1155/2007/75947. (Cited on pages 35 and 38).

[113] C. Wallace, G. Tremblay, and J. N. Amaral. On the tamability of the Location Consistency memory model. In *Proceedings of the International Conference on Parallel and Distributed Processing Techniques and Applications (PDPTA '02)*, volume 3, pages 1542–1550. CSREA Press, 2002. ISBN 1-892512-89-0. (Cited on page 93).

[114] P. T. Wolkotte. *Exploration within the Network-on-Chip Paradigm*. PhD thesis, University of Twente, Enschede, the Netherlands, Jan. 2009. (Cited on pages 15 and 46).

[115] S. C. Woo, M. Ohara, E. Torrie, et al. The SPLASH-2 programs: characterization and methodological considerations. In *Proceedings of the 22nd annual international symposium on Computer architecture (ISCA '95)*, pages 24–36, New York, NY, USA, 1995. ACM. doi: 10.1145/223982.223990. (Cited on page 23).

[116] M.-Y. Wu and W. Shu. An efficient distributed token-based mutual exculsion algorithm with central coordinator. *Journal of Parallel and Distributed Computing*, 62 (10):1602–1613, Oct. 2002. ISSN 0743-7315. doi: 10.1006/jpdc.2002.1847. (Cited on page 64).

[117] Xilinx. Virtex-6 ML605 Evaluation Kit. URL `http://www.xilinx.com/products/boards-and-kits/EK-V6-ML605-G.htm`. (Cited on page 20).

[118] S.-H. Yang, S. Lee, J. Y. Lee, J. Cho, H.-J. Lee, D. Cho, J. Heo, S. Cho, Y. Shin, S. Yun, E. Kim, U. Cho, E. Pyo, M. H. Park, J. C. Son, C. Kim, J. Youn, Y. Chung, S. Park, and S. H. Hwang. A 32nm high-k metal gate application processor with GHz multi-core CPU. In *IEEE International Solid-State Circuits Conference Digest of Technical Papers (ISSCC)*, pages 214–216, Feb. 2012. doi: 10.1109/ISSCC.2012.6176980. (Cited on page 12).

[119] C. Yu and P. Petrov. Distributed and low-power synchronization architecture for embedded multiprocessors. In *Proceedings of the 6th IEEE/ACM/IFIP international conference on Hardware/Software codesign and system synthesis (CODES+ISSS '08)*, pages 73–78, New York, NY, USA, 2008. ACM. ISBN 978-1-60558-470-6. doi: 10.1145/1450135.1450153. (Cited on page 64).

[120] H. Zhang. Service disciplines for guaranteed performance service in packet-switching networks. In *Proceedings of the IEEE*, volume 83, no. 10, pages 1374–1396, 1995. doi: 10.1109/5.469298. (Cited on page 57).

[121] W. Zhu, V. C. Sreedhar, Z. Hu, and G. R. Gao. Synchronization state buffer: supporting efficient fine-grain synchronization on many-core architectures. In *Proceedings of the 34th annual international symposium on Computer architecture (ISCA '07)*, pages 35–45, New York, NY, USA, 2007. ACM. ISBN 978-1-59593-706-3. doi: 10.1145/1250662.1250668. (Cited on page 64).

List of Publications

[JHR:1] P. T. Wolkotte, J. H. Rutgers, P. K. F. Hölzenspies, M. Westmijze, R. Blumink, and G. J. M. Smit. An automated design-flow for FPGA-based sequential simulation. In *Proceedings of the 19th Annual Workshop on Circuits, Systems and Signal Processing (ProRISC), Veldhoven, The Netherlands*, number 2008/14935/STW, pages 126–132. STW, November 2008. URL `http://doc.utwente.nl/65230/`.

[JHR:2] J. H. Rutgers, P. T. Wolkotte, P. K. F. Hölzenspies, J. Kuper, and G. J. M. Smit. An approximate maximum common subgraph algorithm for large digital circuits. In *Proceedings of the 13th Euromicro Conference on Digital System Design: Architectures, Methods and Tools (DSD 2010), Lille, France*, pages 699–705, USA, September 2010. IEEE Computer Society. doi: 10.1109/DSD.2010.29.

[JHR:3] J. H. Rutgers, M. J. G. Bekooij, and G. J. M. Smit. An efficient asymmetric distributed lock for embedded multiprocessor systems. In *Proceedings of the International Conference on Embedded Computer Systems: Architectures, Modeling, and Simulation (IC-SAMOS 2012), Samos, Greece*, pages 176–182, USA, July 2012. IEEE Circuits & Systems Society. doi: 10.1109/SAMOS.2012.6404172.

[JHR:4] J. H. Rutgers, M. J. G. Bekooij, and G. J. M. Smit. Evaluation of a connectionless NoC for a real-time distributed shared memory many-core system. In *Proceedings of the 15th Euromicro Conference on Digital System Design (DSD 2012), Çeşme, Izmir, Turkey*, pages 727–730, USA, September 2012. IEEE Computer Society. doi: 10.1109/DSD.2012.54.

[JHR:5] S. Pande, M. Fearghal, G. J. M. Smit, T. M. Bruintjes, J. H. Rutgers, S. Cawley, J. Harkin, and L. McDaid. Fixed latency on-chip interconnect for hardware spiking neural network architectures. *Parallel Computing*, April 2013. ISSN 0167-8191. doi: 10.1016/j.parco.2013.04.010.

[JHR:6] J. H. Rutgers, M. J. G. Bekooij, and G. J. M. Smit. Portable Memory Consistency for software managed distributed memory in many-core SoC. In *Proceedings of the 20th Reconfigurable Architectures Workshop (RAW 2013), Boston, MA, USA*, pages 212–221, USA, May 2013. IEEE Computer Society. doi: 10.1109/IPDPSW.2013.14. ISBN 978-0-7695-4979-8.

[JHR:7] J. H. Rutgers, M. J. G. Bekooij, and G. J. M. Smit. Programming a multicore architecture without coherency and atomic operations. In *Proceedings of the International Workshop on Programming Models and Applications for Multicores and Manycores (PMAM 2014), Orlando, FL, USA*, pages 29:29–29:38, USA, February 2014. ACM. doi: 10.1145/2560683.2560697. ISBN 978-1-4503-2655-1.

This thesis

```
@phdthesis{rutgers2014:thesis,
    author={Rutgers, Jochem H.},
    title={Programming Models for Many-Core Architectures -- A Co-design
       Approach},
    school={University of Twente},
    address={PO Box 217, 7500 AE  Enschede, The Netherlands},
    year={2014},
    month=may,
    day={14},
    number={CTIT Ph.D. Thesis Series No. 14-292},
    issn={1381-3617},
    isbn={978-90-365-3611-0},
    doi={10.3990/1.9789036536110}
}
```

BibTeX of this thesis

Index

Symbols

$\prec$, 86, 91
$\prec_F$, 89, 92, 96
$\prec_G$, 91, 92
$\prec_L$, 96
$\prec_P$, 88, 92, 96, 129, 131
$\prec_S$, 89, 92, 130
α, 72
β, 58
β-reduction, 114, 124
λ-calculus, 34, 38, 111, 113
λ-term, 113, 119, 120, 126
τ, 59, 60
ψ, 59

A

abstract machine, 35
abstraction layer, 4, 78, 142
acquire, 87, 89, 94, 129
Æthereal, 16, 48, 50, 52, 56, 74
annotations, 78, 84, 94, 125, 129, 132
  entry_ro(), 94, 96
  entry_w(), 130, 132
  entry_x(), 94, 96
  exit_ro(), 94, 96
  exit_w(), 130, 132
  exit_x(), 94, 96
  FENCE(), 125, 130, 132
  fence(), 94, 96
  flush(), 94, 96
application
  coins, 25, 132
  fluidanimate, 24, 61
  parfib, 25, 132
  partak, 25, 132
  prsa, 25, 132
  queens, 25, 132
  radiosity, 23, 51, 57, 61, 100, 102
  raytrace, 24, 51, 57, 61, 100, 102
  skat, 148
  volrend, 24, 51, 57, 61, 100, 102
arbitration point, 55
atomic, 110, 125
atomic-free, 8, 110, 132, 136, 143

B

bakery algorithm, 61
BibTeX of this thesis, 170
black hole, 122
bus, 54, 55

C

C++11, 36, 39, 85
C-HEAP, 22
C99, 36, 39
cache, 14
cache coherency, 15, 17, 30
Cache Consistency, *see also* CC, 82, 93
CC, **82**, 93
characterization, 38
co-design, 2, 7, 31, 137, 142
coins, 25, 132
concurrency, 1, 7, 18, 23, 29, 31, 110, 120
concurrency model, **31**, 137
connection-oriented, 45, 50, 52, 74
connectionless, 46, 53, 74
constraints, 84
contributions, 7, 142
core-to-core communication, 47
core-to-memory communication, 47
CoreMark, 13
currying, 117

D

data race, 111, 124, 130, 136
DDR, *see also* SDRAM
distributed lock, 8, 64, 74, 142
DSM, **17**, 19, 53, 95, 103
DSP, **22**

E

E, 86
easter eggs, *see* somewhere
EC, **83**, 93, 106
embedded systems, 6
endless loop, *see* loop, endless
Entry Consistency, *see also* EC, 83, 93, 106
entry_ro(), 94, 96
entry_w(), 130, 132
entry_x(), 94, 96
execution, 86
execution order, *see also* $\prec$, 91
exit_ro(), 94, 96
exit_w(), 130, 132
exit_x(), 94, 96

F

FCFS, **54**, 57, 65
FENCE(), 125, 130, 132
fence, *see also* $\prec_F$, 89, 94, 125, 130
fence(), 94, 96
FFs, **50**
FIFO, 102
flit, 58
floorplan, 55
fluidanimate, 24, 61
flush, 94
flush(), 94, 96
FPGA, 8, 20, 49
functional programming, 110, 113
future work, 137, 143

G

GC, **119**, **123**, 123, 132, 135
GDO, **92**
GHC, **34**, 132
global λ-term, 122
global order, *see also* $\prec_G$, 91
glue logic, 137
glue tooling, 33, 34, 78
Go, 36
GPO, **92**

H

H, 60
hardware cache coherency, 16, 100, 101
Haskell, 38, 117, 132
Helix, 20, 64, 148

I

I, 59
ILP, **13**, **32**
interconnect, *see also* NoC, 7, 15, 46, 47, 72

J

Java 5.0, 36

K

key insight, 38, 75, 110, 137, 143
KPN, **23**, **31**

L

ℓ, 58, 59
LambdaC++, 115, 132, 143
latency-critical, 48, 58
latency-tolerant, 48
Linux, 19, 22
local λ-term, 122
local order, *see also* $\prec_L$, 88
locality, 17, 23, 135
loop
  endless, *see* endless loop
lossy stack, 125
LUTs, **50**

M

many-core, **4**, 29, 46
MapReduce, 32
memory (consistency) model, 7, **29**, 77, 79, 127
  abstraction, 85, 106
  Cache Consistency, *see also* CC, 82, 93
  Entry Consistency, *see also* EC, 83, 93, 106
  Pipelined RAM, *see also* PRAM, 82, 93
  Portable Memory Consistency, *see* PMC
  Processor Consistency, *see also* PC, 80, 82, 93, 105
  Release Consistency, *see also* RC, 83
  Sequential Consistency, *see also* SC, 30, 80, 93, 105, 128
  Slow Consistency, 82, 89, 92, 96

  Streaming Consistency, 83
  Weak Consistency, *see also* WC, 82
memory bottleneck, 14, 73, 75, 133
memory controller, 52, 62
memory hierarchy, 16, 132
memory operations, 87, 127
memory traffic, 47, 62, 68
memory-to-core communication, 47
message passing, 23
MicroBlaze, 12–14, 19, 49, 72, 95, 96
ML605, 20
MLC, **17**
model of computation, **33**
multi-byte object, 95
multicore, **4**
multicore architectures, 11
mutex, 61, 68, 136
mutex locality, 63, 70

N

n, 54, 58, 72
NoC, *see also* interconnect, **6**, 15, 45, 48, 53, 74
NoFib, 25, 132
non-determinism, 110, 133
NUMA, **4**, 7, 19

O

O, 86

P

P, 86
packet, 55, 58
packet interference, 59
packet interval, 59
par, 120, 122, 125
`parfib`, 25, 132
PARSEC, 24, 61, 74
`partak`, 25, 132
pattern, 86
PC, **80**, 82, 93, 105
performance, 51, 56, 71, 72, 75
Pipelined RAM, *see also* PRAM, 82, 93
platform, 7, **39**, 75, 142
PMC, **8**, **78**, 84, 93, 105, 129, 132, **141**, 142
  annotations, *see* annotations
  back-end, 95, 132
polling, 61, 63
portable, 39, 77
Portable Memory Consistency, *see* PMC
POSIX, 22
posted writes, 48
PRAM, **82**, 93
problem statement, 6, 141
process calculus, 34
Processor Consistency, *see also* PC, 80, 82, 93, 105
program order, *see also* $<_P$, 88
programming model, 2, 6, 7, 18, **35**, 38, 42, 75, 106, 137
  overlap, 38, 137
programming paradigm, 35
`prsa`, 25, 132
Pthread, 18, 22, 31, 36, 39

Q

Q, 59
`queens`, 25, 132
queue, 65, 120

R

`radiosity`, 23, 51, 57, 61, 100, 102
RASP, **33**, 44
`raytrace`, 24, 51, 57, 61, 100, 102
RC, **83**
read-only access, 94
recursion, *see also* recursion
register machine, 33
release, 87, 89, 94
Release Consistency, *see also* RC, 83
relock, 63, 70
ring, 7, 53, 55, 60, 64, 65, 74
RMW, **46**, **60**, 110, 132
RPC, **32**
RTS, **123**

S

S_{bc}, 58
S_{wc}, 58
SANLP, **35**, 38
SC, **30**, **80**, 93, 105, 128
scalability, 72, 75, 135
schedule, 20
SDRAM, *see also* DDR, 61, 62
Sequential Consistency, *see also* SC, 30, 80, 93, 105, 128

service time, 58
shared memory, 15, 16, 23, 110
`skat`, 148
Slow Consistency, 82, 89, 92, 96
SMP, **4**
So long, 175
SoC, **61**
software cache coherency, 17, 95, 100, 132
speedup, 73, 132
SPLASH-2, 23, 51, 61, 74, 100, 102
SPM, **14**, 20, 47, 64, 95, 103, 135
Starburst, 8, 11, 12, 19, 46, 48, 73, 95, 129, 132, 143, 147
streaming application, 23
Streaming Consistency, 83
strong memory model, 30
synchronization, 7, 46, 47, 60, 64, 74, 111
synchronization daemon, 20
synchronization order, *see also* $<_S$, 89
synchronized memory model, 82
synthesis, 50, 56, 65

T

TDM, **16**, 20
thoughts, 9, 43, 107, 145, 151
threading, 18, 31, 75, 110
timestamp, 55, 58
topology, 47, 49, 55
trade-off, 42, 46, 71, 106, 125, 142
transient state, 110
tree, 7, 54, 55, 57, 72
TSO, **17**
Turing machine, 33

U

uniform memory model, 79
utilization, 52, 57, 70, 100, 136

V

V, 86
VLIW, **15**
`volrend`, 24, 51, 57, 61, 100, 102

W

W, 92, 130
Warpfield, 53, 54, 56, 61, 72, 74, 142, 148
WC, 82
Weak Consistency, *see also* WC, 82
weak memory model, 30

KOFF
Programming Models for Many-Core Architectures
A Co-design Approach
Jochem H. Rutgers

ISBN 978-90-365-3611-0

9 789036 536110

Stellingen

behorende bij het proefschrift

Programming Models for Many-Core Architectures
A Co-design Approach

door Jochem H. Rutgers,
te verdedigen op woensdag 14 mei 2014

1 — De efficiëntie van hardware kan alleen bepaald worden, wanneer er een software-kader is gespecificeerd dat beschrijft hoe die hardware goed kan worden gebruikt. *(Hoofdstuk 4)*

2 — Een zwakker geheugenmodel vereenvoudigt de realisatie van cache coherency, maar hoeft het schrijven van een applicatie niet te compliceren. *(Hoofdstuk 5 en 6)*

3 — Het is essentieel om alle platformabstracties gezamenlijk te beschouwen en aan te passen, teneinde vanuit een programmeerperspectief om te kunnen gaan met de huidige trends in many-corearchitecturen. *(Dit proefschrift)*

4 — De kwaliteiten van een goede programmeertaal zijn sterker gerelateerd aan de intuïtie van de mens dan aan de architectuur van de computer.

5 — Hoewel de laatste C-standaard duidelijkheid verschaft omtrent concurrency, rijst er juist onduidelijkheid bij programmeurs of ze wel C schrijven; men kan vrijwel onmogelijk vaststellen of een C-achtig programma voldoet aan de eisen die de standaard voorschrijft om als 'C' aangemerkt te mogen worden.

6 — Bugs zijn vaak het resultaat van de onmacht van de mens om de overweldigende complexiteit van een systeem te kunnen overzien. Desalniettemin betekent vooruitgang in computersystemen het toevoegen van nog meer complexiteit.

7 — C is een populaire programmeertaal, omdat het nauwelijks een programmeerstijl afdwingt, het in principe de programmeur de mogelijkheid geeft om de maximale performance te bereiken, en het de illusie wekt dat programma's portable zijn.

8 — Sociale media danken hun kwalificatie aan de geboden mogelijkheid tot interactie tussen mensen. Dit zijn echter zelden mensen in de directe omgeving, hoewel dat juist degenen zijn met wie sociale interactie gewenst is.