Design and evaluation of a multithreaded many-core architecture

Lankamp, M.

Citation for published version (APA):
Performance improvements for microprocessors have traditionally been achieved by increasing their clock frequency. However, this technique has reached a point where further scaling is impractical.

This thesis describes and evaluates a novel System-on-Chip architecture that focuses on exploiting all forms of concurrency in programs. It does so by defining generic hardware concurrency management extensions in simple multi-threaded cores. These extensions enable low-overhead bulk-creation and dynamic distribution of threads and expose efficient dataflow-like primitives for both inter-thread and intra-thread communication and synchronization.

Additionally, this thesis describes a new cycle-accurate processor architecture software simulator written in C++, in a way to make it a valuable research and education tool by allowing for a clean and relatively high-level description of the architecture.
Design and Evaluation of a Multithreaded Many-core Architecture

ACADEMISCH PROEFSCHRIFT

ter verkrijging van de graad van doctor aan de Universiteit van Amsterdam op gezag van de Rector Magnificus prof. dr. D.C. van den Boom
ten overstaan van een door het College voor Promoties ingestelde commissie, in het openbaar te verdedigen in de Agnietenkapel
op donderdag 17 september 2015, te 12:00 uur

door

Mike Lankamp

geboren te Naarden
The work described in this thesis has been carried out in the Computer Systems Architecture group of the University of Amsterdam, with the financial support of Apple-CORE, a project in the Framework Programme 7 of the European Union, and MICROGRIDS, a project by the Dutch organization for Scientific Research (NWO).

Copyright © 2015 Mike Lankamp.

This work is licensed under the Creative Commons Attribution-NonCommercial 4.0 International License. To view a copy of this license, visit http://creativecommons.org/licenses/by-nc/4.0/.

Author contact: mike.lankamp@gmail.com

Printed by: Gildeprint - Enschede
In memory of my dad, who taught me all he knew about computers...
# Contents

Summary ix  
Samenvatting xi  
Acknowledgments xiii  

1 Introduction 1  
1.1 Instruction-Level Parallelism 2  
1.2 Thread-Level Parallelism 3  
1.3 Parallel Programming 4  
1.4 Contribution 5  
1.5 Organization 7  

2 Related Work 9  
2.1 SIMD 9  
2.2 Instruction Level Parallelism 11  
2.3 Hardware Multithreading 13  
2.4 Dataflow 15  
2.5 Multicore 19  
2.6 Programming Challenges 23  
2.7 Hardware Concurrency Management 24  
2.8 Summary 25  

3 Microthreading 27  
3.1 Families of threads 27  
3.2 Memory 31  
3.3 Places 32  
3.4 Implementations 33  
3.5 Contrast with other programming models 34  
3.6 Summary 35  

4 The Core Architecture 37  
4.1 Fundamentals 37  
4.2 Overview 39  
4.3 Thread & Family Table 39  
4.4 Register file 41  
4.5 Family creation 46
4.6 Family termination ........................................... 52
4.7 Deadlock freedom ........................................... 54
4.8 TLS ........................................................... 56
4.9 Thread scheduler ............................................ 57
4.10 Instruction cache ........................................... 60
4.11 Data cache .................................................. 61
4.12 Choice of ISA ................................................. 63
4.13 Size analysis ................................................ 64
4.14 Challenges and Contributions ............................... 65
4.15 Summary ..................................................... 71

5 The Chip Architecture ........................................... 73
5.1 Microgrid ...................................................... 73
5.2 Places ........................................................... 75
5.3 Network .......................................................... 77
5.4 Memory .......................................................... 78
5.5 External I/O ...................................................... 82
5.6 Size analysis ................................................... 83
5.7 Summary ......................................................... 83

6 The MicroGrid Simulator ......................................... 85
6.1 Emulation vs. Simulation ..................................... 85
6.2 Development principles ...................................... 87
6.3 Design ............................................................. 88
6.4 Correctness ..................................................... 92
6.5 Stalls and deadlock ........................................... 93
6.6 Monitoring ...................................................... 94
6.7 Statistics ........................................................ 99
6.8 Discussion ....................................................... 100
6.9 Summary ......................................................... 100

7 Evaluation .......................................................... 103
7.1 Programming Tools ........................................... 103
7.2 Configuration ................................................... 104
7.3 Numerical kernels ............................................ 104
7.4 Reduction ....................................................... 105
7.5 Cryptography .................................................. 107
7.6 Summary & Discussion ....................................... 109

8 Conclusions & Future Work ...................................... 113
8.1 Discussion ...................................................... 114
8.2 Impact ........................................................... 115
8.3 Future Work ................................................... 116

A Family Table Contents .......................................... 119

B Thread Table Contents .......................................... 121
Summary

Since the 1970s performance increase for microprocessors has been achieved through increases in their clock frequency; the technological advances predicted by Moore’s Law have enabled transistors to become smaller and switch faster. Unfortunately, this is an unscalable solution due to increased heat and power dissipation, energy usage and smaller communication distance that come with increased clock frequencies. Since these advances have already reached a point where further increases are impractical, processor manufactures are forced to steer away from traditional architectural techniques that focused on increasing sequential performance and look to increasing parallel performance.

Given that the traditional methods of obtaining parallelism from serial instruction streams by using techniques such as VLIW, superscalar pipelines and out-of-order execution do not scale well, hardware multi-threading holds the key for scalable performance increase. However, generic multi-threaded programming has been generally seen as too difficult and has therefore been avoided as long as single-threaded performance kept increasing. Now that the latter has stagnated, multi-threaded programming and hardware multi-threading architectures have become a hot topic. Unfortunately, multi-threading architectures generally suffer from a major pitfall: the concurrency is mostly managed in software, creating large overheads for thread creation and thread switching and thus only allowing coarse-grained concurrency to be exploited.

This thesis presents a novel chip architecture that focuses on exploiting all forms of concurrency in programs, fine-grained to coarse-grained, by defining hardware concurrency management extensions which can be applied to any generic von Neumann processing core, which can be composed into a system-on-chip consisting of up to hundreds of these cores and a distributed memory and network-on-chip to connect them together. The architecture consists of modifications to a regular pipelined core that implement ISA extensions and allow for low-overhead bulk-creation and dynamic distribution of threads across a selection of the multi-threaded cores. It defines and exposes efficient dataflow-like primitives for inter-thread communication and synchronization. These primitives are even transparently used to effect dataflow-like synchronization within each single thread, yielding latency tolerance with little to no overhead at runtime. This thesis also describes the required changes to the caches, pipeline and register file and describes the new hardware structures required for the multi-threading book-keeping: the thread table and family table. Finally, it also describes the memory network tailored to a many-core chip with diverse distributed workloads and the network-on-chip protocol which aims to efficiently implement the dynamic distribution of a large set of threads and their synchronization channels across cores.
Additionally, this thesis presents a new cycle-accurate processor architecture software simulator written in C++, which has been used to simulate the aforementioned architecture. The simulator is easily configurable, extensible, diagnosable and inspectable, making it a valuable tool for both architectural research and graduate-level education. It was written to allow for a relatively high-level specification of the architecture, in turn allowing for short design iteration cycles. This high-level specification also allows for easy validation and verification of the architecture.

The simulator comes with external tools which allow the user to select, dump and plot a wide selection of metrics from the simulation, e.g., cycle count, pipeline stalls, number of threads mapped to a core or contention on buses. For the architectural designer it comes with an interactive mode to allow for cycle-by-cycle introspection of the hardware.

Evaluations of the architecture using the simulator show that the performance of the architecture is comparable to contemporary specialized architectures, without losing generality.
Samenvatting

Sinds de jaren zeventig is de prestatieverbetering van microprocessoren bereikt door de frequentie van hun klok te verhogen; de technologische vooruitgangen die voor-speld waren door de Wet van Moore hebben transistoren kleiner laten worden en sneller laten schakelen. Echter is dit een oplossing die niet schaalbaar is wegens het verhoogde energieverbruik, verergerde warmteprobleem en kleinere communicatieafstand die hand in hand gaan met een hogere klokfrequentie. Aangezien deze trend al een punt heeft bereikt waar een verdere verhoging onpraktisch is, zijn proces-sorfabrikanten geformeerd om traditionele architectuele technieken die draaiden om sequentiële prestaties te verbeteren, te verlaten en zich te richten op het verbeteren van parallele prestaties.


Dit proefschrift presenteert een nieuwe chiparchitectuur die zich toespitst op het exploiteren van alle vormen van concurrency in programma’s, van fijn tot grof, door hardware concurrency-beheerextensies te definiëren welke toegepast kunnen worden op elke van Neumann kern, welke op zijn beurt samengesteld kan worden in een system-on-chip bestaande uit honderden van dit soort kernen alsmede een gedistribueerd geheugen en network-on-chip om ze samen te verbinden. De architectuur bestaat uit aanpassingen aan een reguliere gepipelineerde kern die ISA-extensies implementeren en bulk-creatie met een lage overhead en dynamische distributie van threads over een selectie van de multi-threaded kernen toestaan. Het definiert dataflow-achtige primitieven voor inter-thread-communicatie en -synchronisatie en maakt deze beschikbaar voor de software. Deze primitieven worden zelfs transparant gebruikt om dataflow-achtige synchronisatie binnen één enkele thread te bewerkstelligen, waardoor latency getolereerd kan worden met geen tot weinig runtime overhead. Dit proefschrift beschrijft ook de vereiste veranderingen aan de caches, de pipeline en de registers, en beschrijft de nieuwe hardwarestructuren die vereist zijn voor de multi-threading administratie: de thread-tabel en familietabel. Ten slotte beschrijft het ook het ge-
heugennetwerk wat toegespitst is op een veelkernige chip met diverse gedistribueerde werkladingen en het network-on-chip protocol welke probeert de dynamische distributie van een grote set van threads alsmede hun synchronisatiekanalen over kernen te implementeren.

Eveneens presenteert dit proefschrift een nieuwe cycle-nauwkeurige processorarchitectuursoftwaresimulator geschreven in C++, welke is gebruikt om zo’n architectuur te simuleren. De simulator is makkelijk te configureren en is uitbreidbaar, diagnoseerbaar en inspecteerbaar, waardoor het een waardevol gereedschap is voor zowel architectureel onderzoek als universitair onderwijs. Het is geschreven om een specificatie van de architectuur op relatief hoog niveau mogelijk te maken, wat het mogelijk maakt om korte ontwerpiteraties te hebben. Deze hoog-niveau specificatie maakt het ook mogelijk om de architectuur makkelijk te valideren en verifiëren.

Bij de simulator zitten externe tools waarmee de gebruiker een brede selectie van meetpunten van de simulatie kan selecteren, dumpen en plotten, bijv. cycle count, pipeline stalls, aantal threads gemapped op een kern of buscontentie. Voor de architectuurstukker komt het ook met een interactieve modus waarmee het mogelijk is om per cycle de hardware te inspecteren.

Evaluaties van de architectuur met de simulator laten zien dat de prestatie van de architectuur vergelijkbaar is met hedendaagse gespecialiseerde architecturen, zonder generaliteit te verliezen.
First and foremost, I wish to thank my promotor, professor Chris Jesshope; without him I would not have been in a position to do the research and create this thesis: seven years ago I was finishing my Master’s project under his supervision and soon after he offered me a position as a Ph.D. student to expand the work I did during my Master into the realm of proper research. Initially though, I politely declined. I was 24 years old at the time and ready to take my Masters degree and enter industry. It was, however, at his insistence that I realized the opportunity I would leave behind if I declined, so I accepted, and in hindsight I am glad that I did. Fortunately, I could not have wished for a better supervisor. Chris readily let me find my own way and come up with my own ideas; he would always be available for either a rational critique or enthusiastic acceptance of those ideas. During our discussions, he kindly offered guidance and teachings from his own experience. Without doubt I can say that I have thoroughly enjoyed my time as a Ph.D. student under his supervision.

Additionally, my thanks goes out to Andy Pimentel. His honest and unfiltered disbelief at my declination of Chris’ offer as well as his insistence together with Chris’ has been undoubtedly what has finally changed my mind. And now, at the end of the journey, I can honestly thank them both for doing this.

I would also like to especially thank Raphael Poss. Besides being my copromotor, for the majority of time of my research he was my counterpart. Whereas I focused on the hardware side of the group’s research, he voiced the concerns from the language and compiler world. Often, we would end up in front of a whiteboard together, brainstorming and designing, jointly ensuring that the system would run as best as it could yet remain usable for programmers and programming languages. Next to this professional collaboration, it also often felt as if Raphael was the social glue that kept the group close. I fondly remember the movie nights he would organize and the random whatever-the-reason-may-be celebratory cookies that he would bring into the office and share with the group at tea time.

I also wish to thank my other friends and colleagues for making my time at the University of Amsterdam so very enjoyable. Simon Polstra, Mark Thompson and Michael Hicks were always good for a laugh. Michiel van Tol and I probably spent more time reminiscing on the days of old and their video games than we should have. Roberta, Peter and Toktam have also been fantastic people to share an office with. Together, all of them ensured that going to the office never become a chore for me.

My thanks also goes out to Konstantinos Bousias, Thomas Bernard and Zhang Li for founding the work upon which my research was built as well as familiarizing me with the concepts when I started. Without them, too, this thesis would not exist.

Of course, I’d also like to thank my international colleagues from the various
projects we collaborated on for enriching my life with their presence: Bodo, Carl, Dan, Dimitris, Frank, George, Jara, Leoš, Lukas, Martin and Stephan.

And last but not least, I would like to thanks all my other friends and colleagues in the university who have all contributed to making my stay such a memorable one: Merijn, Roy, Roeland, Joe, Vu, Irfan, Clemens, Gokhan, Andrei, Erik, Jian, Qiang, Fangyong and Wei.

Mike Lankamp
Almere, July 2015
Chapter 1

Introduction

“To perform the world’s fastest computation, I divided and evenly distributed the calculations among the 65,536 processors and then squeezed the most performance from each processor.”

- Philip Emeagwali

Since the advent of microprocessors in the 1970s, there has been a continuous striving to increase their performance, that is, to reduce the time it takes to perform a piece of work of a given size. Traditionally, this has been achieved through increases in clock frequency; the technological advances predicted by Moore’s Law [1] in 1964 have enabled transistors to become smaller, allowing them to switch faster. This, in turn, has enabled chip manufacturers to increase the clock frequency, increasing the speed at which instructions are processed, increasing the performance of the processor as a result. However, industry has reached the limits of this technique [2, 3] due to the three major problems of increasing the clock frequency:

• Energy. Due to the nature of transistors, in order to support higher clock frequencies, they must be driven at higher voltages. Consequently, the processor will consume more energy [4]. With the modern-day focus on battery-powered mobile devices and “green computing” [5, 6], frequency increase works against the desired energy efficiency of a processor.

• Heat. Since the dissipated power is (quadratically) proportional to the voltage, the previous point implies increased power dissipation. This power is dissipated in the form of heat, which must be transferred away from the chip to avoid damaging it. Not only does the frequency increase itself generate more heat, but reducing the transistor sizes increases the density of the heat as well. The result of these two factors is that cooling the chip is becoming a greater problem [7].

• Communication. As transistors get smaller and clock frequencies increase, the fraction of the chip that a single clock pulse can travel across in a clock period is reduced [8]. This, in turn, makes monolithic chip design harder, if not impos-
possible, by limiting the size of components or introducing undesired intermediate stages in order to forward data.

These issues indicate that frequency increase is no longer a scalable use of the technology improvements to obtain the desired increases in performance. However, given that the number of transistors on a chip will continue to grow, another method can be used that uses these transistors to obtain performance: *parallelism*—the concept of executing more than one instruction at a time. Any architectural design that accomplishes this must be *scalable*, such that it will allow to provide increases in performance with the continued growth of the number of transistors on a chip. Additionally, any such architecture will also have to deal with the issue of *binary compatibility*: for best commercial uptake it must ensure that the same binary code can execute on a variety of implementations and generations of the architecture.

## 1.1 Instruction-Level Parallelism

Historically, the most successful methods for improving the performance of processors beyond clock frequency scaling have come from exploiting implicit and explicit *instruction-level parallelism* (ILP). With this technique, independent instructions from a serial instruction stream are executed in parallel, thus reducing the time it takes to execute a given program, increasing the performance of the system.

Explicit ILP was implemented in architectures such as VLIW, where the compiler has the responsibility of finding independent instructions and constructing an instruction stream made of fixed-size multi-instruction “packets” which were executed in sequence. Unfortunately, VLIW architectures generally suffered from compatibility problems when newer architectures expand the packet size, and wastage of memory space and bandwidth when the compiler has to pad a packet with NOPs because it cannot find independent instructions. As a result, VLIW architectures were never a commercially successful answer to the problem of stagnating performance for general purpose computers.

The alternative—implicit ILP—was commercially a greater success because it avoided the binary compatibility problems of explicit ILP. A pipeline which can exploit implicit ILP is known as a superscalar pipeline and uses complex logic to analyze dependencies in the instruction stream and schedule multiple instructions in the pipeline in parallel and in a manner that preserves the serial semantics of the instruction stream, thus guaranteeing the same behavior of the processor to its users regardless of its implementation. By increasing the width of the pipeline and using out-of-order execution, superscalar pipelines allow processors to improve the performance of the pipeline well beyond the scaling of clock frequency alone. However, the problem with this technique is that the logic and power required to implement implicit ILP in a pipeline does not scale well with the number of instructions that can be executed in parallel. Additionally, practically there is only a very limited amount of parallel instructions that can be analyzed with this technique.

Instruction-level parallelism is a technique that can offer benefits to single instruction streams and offers binary compatibility, but is not a scalable solution that offers increases in performance into the foreseeable future.
1.2 Thread-Level Parallelism

With the limits of single-threaded performance seemingly reached, processor designers have started to look at multi-threaded designs that can exploit thread-level parallelism (TLP). A multi-threaded architecture is capable of executing multiple independent instructions streams (“threads”) in parallel, either in a single or multiple pipelines. Because the instruction streams are by definition independent, the hardware does not need any expensive logic to find dependencies between instructions; instructions from different threads can always be executed at the same time. Rather than task the hardware with finding independent instructions to execute in parallel, it is now up to the software programmer to ensure that a program is split up into the appropriate threads that can be fed to the processor for execution.

Hardware multithreading allows processor architectures to increase performance in two orthogonal ways. Firstly, it allows multiple execution units or cores to run in parallel on a chip, each executing a thread. Such multicore processors are, design-wise, easily scalable: as the number of transistors on a chip grows, designers can simply fit more and more cores on a chip, allowing it to run more and more threads in parallel. Secondly, more than one thread can be assigned to a single core, allowing a core to hide latency: it can execute instructions from the extra threads when it would otherwise have to stall the pipeline waiting for long-latency instructions to complete. In short, hardware multi-threading offers an architectural solution that can yield an easily scalable solution that yields increases in performance for the foreseeable future.

However, in contrast to architectures exploiting ILP, where a serial instruction stream is parallelized in hardware, multi-threaded architectures that desire to be binary compatible have to deal with potentially more concurrency being defined in software than can be dealt with in hardware. In these cases the concurrency must be serialized in some way. For solutions where the concurrency is entirely defined via software libraries, this serialization can also happen in software. However, solutions that integrate concurrency management in hardware are faced with the challenges of this serialization; this includes the issues related to storing the state of the concurrency which goes beyond the parallelism offered by the architecture.

Additionally, even though many architectures have been created that use hardware multithreading (HMT) even before the limits of instruction-level parallelism had been reached—including the well-known CDC 6000, the HEP, and the Tera MTA—multithreaded architectures have always been commercially less popular than architectures that exploited instruction-level parallelism because programmers used to (and still do) consider splitting up a program into threads (“parallel programming”) hard. The pitfall of multi-threaded architectures is that they only offer a scalable solution to increase performance if the software has enough threads available to keep all of the processor’s cores busy.

---

1 Note that providing a means of communication between these threads is a problem where the solution does not scale as easily.
2 This is the case in most contemporary operating systems, where software threads are interleaved by the operating system onto the available hardware threads.
1.3 Parallel Programming

Even though hardware multi-threading made the hardware scalable, it has not been obvious how to make the software scale along with it. Server applications generally have no problem utilizing the cores because they are *embarrassingly parallel*: a server generally services many clients, each with mostly independent requests which can be processed in parallel. As a result, the server market was readily catered for with multi-core architectures such as Intel’s Xeon and Oracle’s SPARC architectures. General-purpose utilization of multi-core architectures, however, lagged behind; in this case, there is generally only one program active at any time (the program the user is interacting with), and it has to do its work as quickly as possible. It is not trivial for programmers for these kinds of applications to split them up into multiple threads, let alone into enough threads to utilize multi-core architectures in the foreseeable future. In fact, before the limits of instruction-level parallelism and increases in clock frequency were reached, it was easier to program sequentially and let the hardware designers deal with the complexities of extracting parallelism; a vicious cycle that was easier for the hardware designers to follow than to break.

We believe that the problem with traditional parallel programming that causes these difficulties for programmers is that the threads and inter-thread communication and synchronization tend to be coarse-grained. This coarse-grainedness exists because most commercially popular multithreaded architectures leave thread management to software; they define a set number of thread contexts in hardware and leave it up to the software to map the concurrency in the application onto these contexts. This results in large overheads for thread creation and synchronization, which means that there is little benefit to exposing fine-grained concurrency in applications (e.g., small parallel for-loops). This software-based thread management also creates large overheads for inter-thread communication and synchronization.

This disparity between the treatment of threads in software and hardware means that programmers in the current software ecosystem cannot simply express all concurrency in their software; they must pick and choose at what level it is worth it to define concurrency. Because of the existing overheads associated with multi-threading on general purpose computers, this level is often set very high, resulting in programs with limited amounts of exposed concurrency.

Without the availability of enough concurrency in an application, multi-threaded architectures struggle to fully utilize their available resources. As a result, there has been an ever-present atmosphere in the microprocessor world that multi-core and many-core beyond a small number of cores is still not worth it for general purpose computers. It is why microprocessor manufacturers still invest as much as they do into trying to improve the performance of single threads instead of supporting parallel applications, even though the former technique is, ultimately, a dead end.

In the end, this unfortunate state of the software and hardware ecosystems results in a situation where even today’s multi-threaded architectures do not offer a scalable solution for increases in microprocessor performance because, together with the software world, they have not managed to meet the criteria we believe is necessary for success: a scalable architecture supporting fine-grained concurrency in software while offering binary compatibility.
Chapter 1. Introduction

1.4 Contribution

This thesis presents an architectural design that attempts to solve the problems that have been outlined above. In order to achieve a scalable future-proof architecture that supports fine-grained concurrency in software while offering binary compatibility, we identify two major design goals:

1. Concurrency management should be moved from software to hardware: applications should have a direct and low-overhead mechanism in the instruction set to define concurrency and synchronization. This ensures that programmers can express the fullest amount of concurrency in a program without regard for the hardware limitations. It is the hardware’s responsibility to curb the concurrency defined by the program to fit the amount of parallelism offered by any particular hardware implementation. This ensures the future scalability of software as the hardware evolves.

2. Traditional, complex and unscalable hardware logic for increasing the throughput of single threads should be removed and replaced by a simpler dataflow management of threads at the level of individual instructions. Although this sacrifices the performance of single threads, we believe that the resulting simplification of the cores means that more cores can fit on a chip, allowing for additional parallelism to execute the concurrency exposed in the previous point. The focus of this approach is thus on increasing the performance of parallel programs.

Therefore, the main research question that we aim to answer in this thesis is:

“is such an architecture feasible and viable and if so, what are the necessary components for such an architecture?”

We aim to answer this question in this thesis by presenting:

- the general design of a single-core architecture and its components, capable of fast, interleaved execution of multiple threads with dataflow scheduling, and efficient, dynamic distribution of software-defined concurrency across available hardware threads. (Presented in chapter 4 and in publications P.1 P.4 P.6 P.7 P.8 P.10 and P.11)

- the composition of this single core into a larger system-on-chip while ensuring that the composition is scalable; additional cores can be added without added complexity to the logic. (Presented in chapter 5 and publications P.1 P.4 P.6 P.7 P.8 and P.11)

---

3In this thesis, “we”, “our”, and “us” are used to reflect the close collaboration between the author of this thesis and his promotor during the research presented in this thesis. However, all the contributions, design, implementation, empirical evaluation, and other research elements included in this thesis have been performed by the author himself.
Chapter 1. Introduction

• the validation of the design by means of the secondary result of this thesis: an operational and verifiable cycle-accurate software emulation of a multi-core chip implementing the architectural design. (Presented in chapters 6 and 7 and publications P.1, P.2, P.3, P.4, P.5, P.6, P.7, P.8, P.10, P.11 and P.12)

Additionally, the design takes into account the key challenges that appear when designing such an architecture:

• How to ensure that the design is efficient? To achieve the best performance, the design should be as efficient as possible to avoid wasting time, which decreases performance. Yet, speculation and busy waiting should be avoided to conserve energy. This topic is covered in chapter 4.

• How to ensure that the design is scalable? It should be possible for the design to scale upwards with the continued growth of transistors on a chip and still support the concurrency defined in software. This topic is covered in chapters 4 and 5.

• How to ensure that the software is forwards and backwards compatible? For commercial viability, software should not have to be recompiled to be able to run on newer generations of the architecture. This topic is covered in chapter 4.

• How to ensure that the management of concurrency in hardware, complete with the necessary communication between cores in distributing the work is fair and free of deadlock. This topic is covered in chapters 4 and 5.

The design presented in this thesis is the general design of a scalable, energy-efficient and easily programmable many-core processor using hardware multithreading with hardware concurrency management and fast thread switching. The execution model of this architecture uses bulk creation of threads with constrained, but efficient inter-thread communication. The architecture design has been made using both new and existing techniques by combining them in such a way as to achieve the goals of scalability and energy-efficiency. The bulk creation of threads from the execution model is directly supported in the architecture, which also performs the scheduling and mapping of those threads. A large register file with dataflow synchronization implements the intra- and inter-thread communication and synchronization. Long-latency operations are implemented as split-phase operations in order to provide latency tolerance.

The design is a general design by virtue of being applicable to any kind of RISC-like architecture, but has been fleshed out into a particular multi-core system based on the DEC Alpha instruction set for evaluation. This instruction set embodies the RISC concepts in a clean and easily implementable instruction set, allowing the research to focus on the core issues regarding multithreading. As such, the architectural components described in this thesis should be considered as a set of components and guidelines that can be applied to other architectures to achieve the same benefits.
1.5 Organization

This thesis is organized as follows. Chapter 2 performs a survey of the existing techniques and architectures to exploit parallelism. Chapter 3 explains Microthreading, the generic parallel programming model that is the architecture’s main programming model and was co-designed alongside the architecture.

Chapters 4 and 5 describe the main result of this thesis: the design of the architecture mentioned in the previous section. In particular, chapter 4 focuses on the architecture of a single core and its components. It describes why such a particular design has been chosen and what problems and design choices led to the design in its current form. Chapter 5 describes how this single core architecture can be composed into a System-on-Chip with accompanying systems such as memory and I/O to achieve a scalable solution.

Chapter 6 describes the software simulation framework that has been developed in order to test the designed architecture and chapter 7 shows, by experimental validation using this simulator, how the designed architecture meets the goals of performance, scalability and ease of programmability. Together, all these chapters answer the primary research question of this thesis by showing that the architecture is feasible, viable and successfully meets the challenges laid out above.

Finally, to conclude this thesis, chapter 8 will summarize the work described in it and place it in perspective of the current architectural developments.
Chapter 2

Related Work

“You have to know the past to understand the present.”
- Carl Sagan

In the past, there have been many approaches to increase the performance of processors by increasing their parallelism. This chapter provides a brief survey of the past and ongoing work related to such architectures. It will describe SIMD architectures, where data parallelism is obtained by duplicating the processor’s execution units so the same instruction can act on multiple pieces of data. After this, it will describe architectures that implement some form of instruction-level parallelism, a technique which enables processors to execute independent instructions from an instruction stream in parallel. The next level of parallelism is thread-level parallelism, as implemented by techniques collectively known as hardware multithreading (HMT). Here, processors have execution states for multiple independent instruction streams and use these to improve performance where the previous techniques could not. Lastly, an entirely different form of parallelism is discussed: dataflow. A complete change from the von Neumann style of architectures, dataflow architectures offer maximum parallelism, but are not easy to implement because it is hard to efficiently manage all the exposed parallelism.

The final sections of this chapter will discuss the issues and challenges of multicore processors; they are a trend that has become inevitable considering Moore’s Law and the issues discussed in the previous chapter.

2.1 SIMD

One of the earliest approaches to improve performance of traditional von Neumann-style architectures was to increase the data width in the processor. By having a single instruction stream operate on multiple pieces of data, the throughput of applications could be improved without requiring additional instruction streams or cores. This technique is collectively known as Single Instruction, Multiple Data (SIMD), which was first used in the Illiac IV[12], among others. The following subsections highlight recent successful implementations of SIMD architectures.
Chapter 2. Related Work

2.1.1 SSE/MMX

Streaming SIMD Extensions, or SSE for short, was introduced by Intel as an extension to their hugely popular x86 instruction set \[13\]. It was designed to replace their earlier attempt at introducing SIMD support to their Pentium line of processors, MMX, and has since been extended with additional instructions to SSE4. SSE added new 128-bit vector registers to the processor that can be interpreted as four 32-bit or two 64-bit floating point values. The instructions operate on the values in parallel.

SSE and MMX are aimed at increasing the performance of otherwise sequential applications that contain trivially parallelizable operations, such as multimedia or signal processing applications, which typically contain vector operations such as vector addition or subtraction. As such, SSE is only of limited use, in highly specific areas, and only offers limited speedup.

2.1.2 CUDA

Compute Unified Device Architecture (CUDA) is a parallel computing API developed by Nvidia \[14\]. It allows programs to use Nvidia’s graphics processing units (GPUs) to run parallel computations, by taking advantage of the fact that GPUs are highly efficient SIMD vector processing units, consisting of multiple identical cores, executing the same code in lock-step. This means that CUDA is only applicable to applications that match the SIMD model.

As a consequence of the lock-step implementation, branches are implemented with predicated execution, which wastes energy as all paths of the branch need to be executed on all processing units if there is divergence in the control flow among the processing units. CUDA partially mitigates this problem by splitting up its cores into groups called Streaming Multiprocessors (SMs). Each SM is an independent SIMD processor that is capable of executing a single thread definition in lock-step on its SIMD cores.

CUDA shares some aspects with Microthreading (see chapter \[3\]) in that the programmer provides a single thread definition which is instantiated several times in hardware. Unlike Microthreading, however, the threads in CUDA run in lock-step in a true SIMD fashion.

Additionally, CUDA is an accelerator interface; this means that CUDA devices are in essence accelerators connected to traditional single or multi-core architectures. As such, there is a non-trivial amount of work involved with sending work to a CUDA device, which results in CUDA only being applicable for large uniform data-parallel operations that are big enough to offset the overhead of initiating the work. Because the CUDA architecture is different from the host architecture, a program designed to make use of CUDA has to be manually separated into the parts that will run on the host, and the parts that will run on the GPU, both using different languages.

---

1 because all cores execute the same code, CUDA can also be seen as an SPMD implementation.
2.2 Instruction Level Parallelism

If SIMD cannot be used to achieve data parallelism, the performance of processors can be increased by executing multiple instructions (performing different operations on different data) in parallel to exploit instruction-level parallelism (ILP). This is the name for a broad collection of techniques whereby instructions from a single, serial instruction stream are executed in parallel by the hardware. ILP is different from thread-level parallelism (TLP) in that TLP has multiple instructions streams (“threads”) explicitly defined by software whereas ILP has only one.

The motivations for exploiting ILP were two-fold: on the one hand it could increase the number of instructions executed per cycle (IPC). On the other hand, it allowed the processor to tolerate long memory latencies, which have been growing as a result of the so-called “memory wall”; this term describes the disparity in clock frequency increase between processors and DRAM [15]. The result of this disparity is that as processor clock frequencies increase, accesses to memory take longer and longer from the processor’s point of view. These latencies are unavoidable, but can be tolerated with parallelism. While long-latency operations such as memory accesses are waiting for their result, the processor can execute independent instructions while the memory access is pending. Without exploiting ILP at the least, this would not be possible.

2.2.1 Superscalar

Superscalar processors are processors that are capable of issuing and retiring more than one instruction per cycle. Although strictly speaking the term describes any architecture capable of executing more than one instruction at the same time, such as VLIW processors described below, traditionally the term has been used to denote architectures that find and execute independent instructions from a (by definition) serial instruction stream. That is, the instruction stream contains no explicit information about instruction-level parallelism and the superscalar processor has to find concurrent instructions and dispatch them in parallel. The benefit of this approach is backward compatibility: because the semantics for the instruction stream do not change over the various implementations of the architecture, every implementation can execute code for previous architectures. However, the cost of this approach, aside from the necessary duplication of hardware, is the logic required to find independent instructions from a sequential instruction stream and schedule those instructions into the pipeline such that no data hazards occur. Because the complexity of this logic is quadratic at best and exponential at worst with respect to the number of instructions executed every cycle [16], this approach does not scale. Additionally, serial instruction streams tend to have a limited amount of ILP that can be exploited [17, 18].

Another technique to further exploit ILP in superscalar architectures is to allow instructions in the pipeline to overtake those issued earlier in order to avoid stalling the pipeline as a whole when long-latency operations such as memory loads are held up in the pipeline. This is called out-of-order execution. This technique is often used in a superscalar processor as the nature of a superscalar pipeline makes it desirable for instructions to complete in an order different from the one they were issued in (in-order issue). It’s also possible to issue the instructions in a different order as well (out-of-
order issue), although this technique is much more costly in terms of hardware\cite{16}. The downside of out-of-order execution is the substantial amount of non-linear logic and energy required for the dependency analysis \cite{19} that determines if instructions can overtake each other, as well as reorder buffers for in-order completion, which is required to support precise exceptions.

\subsection{VLIW}

Very Large Instruction Word architectures execute instruction words which contain multiple independent instructions packed side-by-side \cite{20}. It is, in essence, a superscalar architecture where the data dependency analysis is performed by the compiler instead of the hardware. As a form of explicit instruction-level parallelism, the benefit of VLIW is that the non-scaling logic for extracting ILP from the instruction stream is no longer required. However, the problem with this technique is that the compiler must pack independent instructions into instruction words. If such instructions cannot be found during compilation, NOPs must be inserted into the instruction words instead, resulting in large amounts of wasted space in code memory and unnecessary energy dissipation by unneeded but active functional units. This will cause an increase in I-Cache misses, which, combined with the memory wall, degrades performance \cite{21}. Although multiple solutions in the form of code compression \cite{22,23,24,25,26} have been suggested, by their nature they all require decoder logic and decoder tables, which consume both power and space.

Additionally, VLIW mandates a static scheduling. The compiler requires knowledge about the delays of instructions on the target architecture, and must schedule the instructions in such a way that none are held up because their dependencies are not available yet. This works for instructions with known and deterministic delays, but this breaks down, for instance, with memory loads. The delay of a memory load in a typical multi-level memory system depends on the availability of the data in the caches, and cannot be known at compile time in a general-purpose system due to unknown applications sharing the system. Since VLIW architectures lack hardware for out-of-order execution, the pipeline must stall on a cache miss, wasting time and energy.

Finally, VLIW has a binary compatibility issue; if future architectures change to include more execution units such that more instructions can be execution at the same time, the syntax of the instruction stream has to change, which prevents previous binaries from executing on the new architecture.

These issues mean that VLIW is best suited for use in embedded systems, where a single architecture is targeted and the applications running it are known in advance. In this case, memory loads can be bound by a reasonable worst-case latency, which in turn makes it easier for the compiler to make an optimal scheduling that prevents stalls. The compatibility issue is also not a problem, since the applications are compiled for the single embedded architecture.
2.2.3 EPIC

Explicitly Parallel Instruction Computing (EPIC) is an architecture developed by Intel and HP and implemented in their IA-64, Itanium and Itanium 2 processors [27, 28]. It is a VLIW-based architecture, where the compiler packs multiple instructions into instruction words. However, EPIC aimed to improve upon VLIW by using techniques found in superscalar processors. It uses speculative or predicated execution to deal with branches and tolerates memory latency by making the compiler control the caching. The latter can issue prefetch commands and even decide placement of data in caches. This gives the compiler additional control over memory latencies, allowing it to adjust the static schedule of instructions accordingly.

EPIC solves VLIW’s binary compatibility problem by allowing the processor to fetch and issue instructions packs (called bundles) in parallel, much like superscalar pipelines can issue single instructions in parallel. Unfortunately, despite its improvements, EPIC still has problems. Firstly, the instruction stream still requires inserted NOPs, increasing the code size. Secondly, EPIC’s fixes over VLIW such as predicated execution and super-scalar execution are wasteful of energy and clock cycles, just as in a non-VLIW pipeline, and its speculative execution comes at the cost of power and area.

2.3 Hardware Multithreading

Since extracting parallelism from a single instruction stream is not a scalable solution to the performance problem, the next logical step is to have not one, but multiple instruction streams, called “threads”. If a program defines enough threads, performance increase can be obtained from executing these threads in parallel. Notably, threads allow processors to better tolerate long-latency instructions such as I/O or memory loads that miss the cache: the typical latencies for these operations exceed the size of the window in which conventional ILP solutions operate. This so-called thread-level parallelism has been studied before [29], but is harder to program since, unlike ILP, the programmer must now deal with the semantics of instructions running in parallel.

2.3.1 Barrel processors

A barrel processor is any type of processor which supports multiple threads and interleaves their execution cycle-by-cycle in the pipeline. This means that each thread only runs one instruction every $N$ cycles, where $N$ is the length of the pipeline. The biggest downside to barrel processors is that full pipeline utilization is only reached when there are as many threads as pipeline stages. However, even with a fully saturated pipeline, barrel processors still execute only one instruction per cycle, so under ideal conditions they perform worse than superscalar architectures. The advantage of barrel processors appears when there are enough threads to keep the pipeline saturated, whereas in a single-threaded pipeline, various hazards between instructions within the single thread can cause stalls. Additionally, because every stage of the pipeline in a barrel processor will be executing an instruction from a different thread,
there will no need for pipeline bypasses, simplifying the design and potentially allowing the processor to be clocked at a higher frequency.

Examples of barrel processors are the CDC 6000[9], the HEP[10] and the Tera MTA[11].

### 2.3.2 Hyper-threading

Designed by Intel as an extension to their Pentium processors, Hyper-Threading[30] can be seen in two ways. It can be seen as the merging of two cores into one, or as a duplication of certain pipeline logic. Hyper-threading allows two threads to simultaneously execute on a single core by duplicating the thread state resources (e.g., registers). Having two independent thread contexts available allows a superscalar pipeline greater freedom in selecting instructions that will prevent pipeline stalls. Whereas a dual-core solution theoretically offers a guaranteed speedup factor of 2 (memory contention left aside), hyper-threading does not. This is because the execution units in the pipeline are still shared between the two threads that are running; in other words, a pipeline running two threads, each maximally utilizing the superscalar pipeline, will benefit least. However, common situations where the results of a memory load stalls the rest of the pipeline can be mitigated by running non-memory instructions from the other thread on the pipeline. Because the design of out-of-order superscalar pipelines already have the required bookkeeping to keep track of such dependencies, this is a relatively cheap extension to the architecture.

### 2.3.3 HEP

The HEP [10] (Heterogenous Element Processor) is a parallel MIMD processor developed in 1980s that consisted of 16 processors (or PEMs, Process Execution Modules, as they were called) where each core could execute 64 threads concurrently by interleaving them cycle-by-cycle in its single 8-stage pipeline. The factor of 8 additional threads are there to allow the core to tolerate memory access latencies, created by the switching network that connects the cores to the external memory.

Synchronization between running threads occurs in memory via an extra bit per word. This synchronization bit can be used by certain instructions to achieve blocking read and write behavior to empty and full memory locations, respectively. Two programming language extensions were added to exploit the hardware features. First, a CREATE instruction, analogous to CALL, except the calling thread continues while the callee executes concurrently. Secondly, asynchronous variables that map to the in-memory I-structures mentioned above.

These extensions allow programs to efficiently create a pool of threads and schedule them using the asynchronous variables. The example in [10] has a procedure to create 12 threads that each operate on one element in an array, and use an asynchronous variable to synchronize the workload between them, looping until all \( N \) elements in the array have been processed, after which the dispatching procedure completes.

Although innovative, the implementation of the HEP does not scale to modern architectures. In-memory synchronization only becomes slower as memory latencies
increase and locality is not exploited. Nonetheless, as chapter 4 will illustrate, the architecture described in this thesis shares a common ideology with the HEP.

2.3.4 Oracle SPARC

The SPARC family of systems, developed by Oracle, is a family of multicore multi-threaded systems[31]. Starting with the UltraSparc T1 (also known as Niagara [32]) and UltraSparc T2 (also knows as Niagara 2 [33]), the latest version is the SPARC T5, which consists of 16 cores, where each core supports executing 8 threads in a hyperthreaded pipeline. It can run legacy applications and effectively functions like an SMP machine and has the downsides of such architectures.

Threads are created through libraries in combination with the operating system and synchronize through the shared memory, resulting in large overheads for creation and synchronization. As a result of this design, the Niagara 2 mainly targets internet servers and related systems that have large independent workloads which are suited to the architecture. Despite having more hardware threads than other contemporary processors, it still relies on software to schedule the concurrency in applications onto the hardware threads. This makes the architecture undesirable for exploiting fine-grained concurrency.

2.3.5 Intel SCC

Intel’s Single-chip Cloud Computer [34], developed as part of Intel’s Tera-Scale computing research program [35], is a processor consisting of 48 Pentium cores connected with a mesh network and on-chip message passing network for inter-thread communication. It also has a non-shared memory where software is responsible for cache coherency. It was designed as a research platform to study parallel programming and has been described as a “cluster on a chip”. The problem with the SCC is that, like other multicore architectures, it targets coarse-grain concurrency, and disregards fine-grain concurrency. Neither does it support multiple threads on a single core, which would allow for energy efficient techniques such as latency hiding.

2.4 Dataflow

Traditional processors follow the so-called von Neumann execution model. In this execution model the processor executes a list of instructions one by one, the current state identified by a program counter. Instructions specify a location in one or more memories where outputs are stored and inputs are read from. In contrast to the von Neumann model is the dataflow model [36], where execution is driven by the availability of data, rather than a list of instructions. In the pure dataflow model, each instruction contains the address of the instruction(s) that will receive its output. There is no explicit order in execution, only the one implied by the dependencies of instructions.

Theoretically, dataflow allows for maximum parallelism on an instruction-by-instruction basis. However, a von Neumann model captures its current state efficiently with a single program counter. With dataflow, the current state of a program consists
of every currently activated instruction and all their data (called tokens). Because the state of dataflow programs can be so large, pure dataflow architectures are hard to implement. Instead, architectures have been developed that are a mix between the dataflow and von Neumann execution models. At their core, they execute sequential instruction streams, taking advantage of the small state of the von Neumann model. However, they are extended with dataflow semantics on some scale, allowing for a more dynamic execution based on data dependency, allowing them to better exploit parallelism in programs.

Listed below are some key architectures that have implemented the dataflow model in full or used it as an extension to the von Neumann model.

2.4.1 Monsoon

Monsoon is a realization of an explicit tagged-token dataflow architecture, developed by MIT [37]. By using an explicit token store (ETS) [38], it mitigates a lot of the inefficiencies of implicit token stores of pure dataflow, because implicit token stores are primarily implemented via associative memory, whereas explicit token stores are able to use directly addressed memory; the former is more expensive to implement in terms of area, access time and power than the latter.

In ETS an activation frame is allocated in memory whenever a code block (e.g., function or loop iteration) is started. The activation frame is big enough to contain all tokens that can be generated in the code block, which can be calculated by the compiler. Instructions in the code block specify an offset within the activation frame as their input tokens. The Monsoon has a large synchronization namespace, inexpensive synchronization and is able tolerate memory and communication latency. To achieve the latter, it uses split-phase memory operations with I-structure synchronizers.

One of the problems of the Monsoon stems from its fixed interleaved pipeline: a single core, with an 8-stage pipeline, can only be fully utilized if there is at least 8-fold parallelism. [39].

2.4.2 Wavescalar

Wavescalar [40] is a pure data flow architecture that uses data tags, called wave numbers, to differentiate between the data from multiple loop instances and function invocations. It also uses these wave numbers and explicitly annotated memory instructions to order memory requests and thus enforce sequential memory semantics. The instructions are cached on the processing elements themselves and send their results directly to their consumer instructions on the same or nearby processing elements. Threads are supported by extending the data tag with a thread ID to differentiate data from different threads.

Where instructions are placed on a grid of processing elements is worked out by both the compiler and hardware. The compiler groups dependent instructions into segments which will be placed on a single processing element by the hardware. Subsequent segments are placed on nearby processing-elements by the hardware with a simple fill algorithm.
Chapter 2. Related Work

The disadvantage of Wavescalar lies in the fact that it severely constrains memory operations of threads into a sequential ordering. This constraint allows the processor to utilize only limited amounts of the concurrency encoded in the instruction stream. Another disadvantage is one that is inherent to most pure-dataflow architectures: if there is a sudden “explosion” of concurrency in the program, all of those states must be captured and stored. With limited buffer and queue size, thrashing and stalling become dominant bottlenecks.

2.4.3 P-RISC

P-RISC, or Parallel RISC [41], aims to create a hybrid dataflow/von Neumann machine where instead of synchronizing on individual instructions, dataflow synchronization is used to synchronize on sections of sequential code. It introduces new instructions to fork and join threads and supports multiple outstanding split-phase memory operations that can return in any order. One of the problems with the P-RISC approach is identical to the HEP’s: it time-shares the pipeline by interleaving threads on every cycle. This requires at least as many threads as there are pipeline stages to keep the pipeline completely busy.

2.4.4 Multiscalar

A Multiscalar processor [42] works by having the compiler split up a program into chunks called “tasks” and executing these tasks concurrently on different processing elements. The processing elements, which are connected with a unidirectional ring network, operate like simple in-order pipelined processors. Tasks can be different pieces of code or different instances of the same piece of code, such as iterations in a loop. The main instruction stream is annotated with a secondary instruction stream, which contains information on register synchronization and task termination for every instruction in the main instruction stream. The register synchronization bits are set by the compiler to inform the processor when to send register values from one task to the next. The processing element containing the task that reads a shared register will stall until that register has been written by the previous thread. Synchronization between tasks via memory to ensure sequential semantics is done by monitoring past and outstanding loads and stores, and squashing a thread if it turns out it loaded a value before a thread preceding it had written it. Tasks that represent iterations are speculatively dispatched to processing elements and excess tasks are squashed when the loop exit condition becomes known.

Multiscalar has, at a first glance, many similarities with the techniques presented later in this thesis in that with both architectures the compiler must separate the program into blocks of annotated code which can run concurrently on multiple simple, in-order pipelined processors. The difference, however, lies in the fact that with Multiscalar, only a single task can execute on a processing element at a time. If, because of a data dependency, the task is forced to stall, the entire processing element is wasting cycles. Microthreading, as will be described in chapter 3, can continue executing instructions from other threads mapped to the same processor. Additionally, Multiscalar has no explicit synchronization on memory and its run-time per-address
data dependency analysis is a form of speculative execution, wasting time and energy when it squashes tasks.

2.4.5 Inthreads

Inthreads \cite{43} is an architecture that provides explicit synchronization primitives and instructions in the form of binary semaphores and wait, signal and clear instructions that act upon them. The multiple-issue pipeline dedicates each fetch stage to a single thread and has a special stage that stalls threads when it waits on a semaphore that isn’t signaled. Once another thread signals the semaphore the pipeline resumes executing from the stalled thread. Once a thread has passed this wait stage, it is processed by a normal out-of-order pipeline with a shared register space and possibly shared functional units.

The compiler has to explicitly create threads, designate thread identifiers and map the thread’s registers in the shared register space such that private registers of concurrent threads don’t overlap. Communication between threads is done by the compiler mapping shared registers to the same location in the shared register file and inserting proper semaphore instructions to effect synchronization.

Although Inthreads offers low-cost synchronization primitives, the maximum concurrency is still limited by the issue width of the pipeline and when a thread is stalled on a semaphore, no other thread can be run in its place. The compiler also needs intricate knowledge of the hardware, adjusting the number of threads to the issue width of the pipeline. This could hinder backwards or forward compatibility. The threads themselves are relatively long-lived, compared to the Micro-threading model. They are used mainly to easily balance the load of a loop across the width of the pipeline. This load balancing has to be hardcoded by the compiler and there seems no way to dynamically adjust this depending on available resources or hardware structure.

2.4.6 Scheduled Dataflow

Scheduled Dataflow \cite{44,45} aims to solve the memory latency problem by splitting threads up into three phases: loads, execution and stores. A single SDF processor consists of two sub-processors, called the Synchronization Processor (SP) and the Execution Processor (EP). The former performs the loads and stores at the start and end of a thread and the latter executes the main body of the thread. The reasoning here is that a thread will not be scheduled onto the EP until its memory loads have been completed by the SP. This then allows the thread to be run in a non-blocking mode by the EP. However, there are several downsides with this approach. First, the SP does not use split-phase memory operations, effectively pushing all memory latency problems to the SP. Secondly, the model does not allow for loads or stores to occur in the body of the thread. For threads with data-dependent control flow, this means speculatively hoisting all memory loads up to the start of thread. The SP will then be busy loading data which might not even be used, wasting both bandwidth and energy.
2.4.7 EDGE

Explicit Data Graph Execution (EDGE) is an architecture that has the compiler explicitly encode a data graph in the compiled code. The EDGE architecture has been implemented in TRIPS at the University of Texas [46]. The compiler segments sequential code into hyperblocks, which consist of 128 dataflow-executed instructions. The result of each instruction is sent directly to its consumers within the hyperblock, leaving the register file just to store results between hyperblocks. TRIPS features 16 execution cores per processor, with multiple processors on a chip. Hyperblocks are mapped to cores such that each core in a processor has at most 8 instructions mapped to it. TRIPS sequentially and speculatively executes hyperblocks on a chip’s processor.

In effect, TRIPS, and the EDGE architecture, consists of multiple 16-wide out-of-order superscalar processors on a chip that can hold up to 1024 instructions in flight. However, EDGE is only a dataflow model inside of hyperblocks. The hyperblocks themselves are still sequentially and speculatively executed. As a result, this model is only scalable up to the size of the hyperblocks. For increased scalability, the hyperblocks must be enlarged, which prevents previously compiled code from being run on the new hardware, limiting binary compatibility.

2.4.8 DDM-CMP

A Data-Driven Multithreading Chip Multiprocessor [47] uses a conventional off-the-shelf processor with a Thread Synchronization Unit (TSU) attached to them for dataflow synchronization of threads. It is a form of pure dataflow, but on thread level rather than instruction level. Threads have inputs and outputs and threads are scheduled according to the availability of their code and input data in the cache. Threads are defined by the compiler by splitting up functions and loop iterations into basic blocks. As such, it exploits both coarse and fine-grained concurrency. It solves the problem of lack of locality with respect to the I-cache that is inherent to dataflow models by prefetching cache-lines in the TSU when threads are woken up, but before they are executed by the attached processor.

A downside of this approach is the limited benefit for existing binaries; the latency-hiding of memory operations only works for explicitly defined threads with explicitly defined dependencies. Existing binaries for the target ISA do not receive any benefit at all. Furthermore, since the TSU is a simply a thread manager for a regular processor, the processor itself is not multi-threaded. This prevents the processor from using threads to hide latency for non-memory operations such as floating-point operations. It would have to rely on traditional energy-inefficient techniques such as out-of-order execution to hide these latencies.

2.5 Multicore

Traditionally, the additional space on a chip has been used to increase the on-chip cache memory. However, since such a monolithic chip design is not scalable as noted in chapter 1 and earlier in this chapter, processor manufacturers are instead using the
Chapter 2. Related Work

Figure 2.1: Design spectrum for core size. On the one end, few large cores for sequential performance. On the other, many small cores for parallel performance.

Additionally, when a chip contains multiple cores, there is now a design spectrum of how much space to allocate to each core. When the cores are small, many of them fit on a chip, which allows for greater performance for independent tasks (i.e., parallelism). On the other hand, large cores offer greater performance for a single task by incorporating expensive logic such as speculation (branch prediction or cache prefetching) or support for instruction-level parallelism. This spectrum can be viewed differently by noting that large cores are characterized by a high IPS/core, but low IPS/chip and lower IPW due to the energy inefficient methods by which they achieve their performance, whereas the smaller cores are characterized by a low IPS/core, but high IPS/chip and higher IPW. This spectrum is illustrated in figure 2.1.

The argument against fully sequential chip design is non-scalability and energy usage. The extensions which improve a processor’s performance for sequential execution do not scale well in either hardware or energy. Support for dynamically extracting parallelism from a serial instruction stream (also known as instruction-level parallelism) requires an amount of hardware that grows exponentially with the instruction window size. Branch prediction aims to reduce the number of pipeline flushes at the cost of power and chip area required for the prediction logic and branch history data. Although branch predictors can be very accurate at around 95% accuracy, they can by definition never be perfect and a wrong prediction still results in a pipeline flush. Cache prefetching tries to avoid the delay of a cache-miss by looking at the memory access patterns of a program and guessing what cache-line they will access next. Although the benefit can be a substantial gain in performance, there is a high chance for prefetch wastage (spending energy and memory bandwidth

---

2IPS: Instructions executed per Second
3IPW: Instructions executed per Watt
fetching data from off-chip that will not be used) and cache pollution (a prefetched cache-line evicting a required cache-line).

On the other end, fully parallel chip design is discouraged by Amdahl’s Law\footnote{Amdahl’s law roughly states that by improving parallel performance, a program can at best become as fast as it’s non-parallel part.} since most non-trivial programs will contain sections of non-parallelizable code. When the parallel sections are sped up by distributing them across the many cores on the chip, the sequential section will quickly dominate the total runtime of the program. Additionally, code with complex data dependencies, even if parallelizable, can actually suffer from being distributed across multiple cores, because locality of data usage is lost. Thus, there will always be an argument for cores optimized for executing sequential code. However, the question now becomes one of focus. Should a chip be mainly focused on executing sequential code, with some acceleration for parallel section, or vice versa? For the best scalability, programs should be written as concurrently as possible. This is reflected by Gustafson’s Law\footnote{Gustafson’s law roughly states that by improving parallel performance, a program where the parallel part dominates the non-parallel part can perform more work in the same time.} Combined with the fact that general-purpose processors can run different independent applications in parallel as well, the focus of the chip should definitely lie with parallel performance.

### 2.5.1 Heterogeneity

With differently-sized cores available to a chip designer, another spectrum opens up: how many of each to put on the chip. On the extremes of the spectrum, the chip is dedicated entirely to large cores, or entirely to small cores. The optimal location in this spectrum depends on the expected use of the chip. With most area dedicated to large cores, the focus of the chip lies with sequential performance. At the other end, many small cores indicate the chip focuses on parallel performance. Interesting situations arise when moving away from the extremes. Allocating some area to smaller cores next to a few very large cores keeps the focus on sequential performance, but makes the small cores available as “accelerators“ for highly concurrent sections of code that can take advantage of being distributed across multiple cores. Conversely, allocating some area to larger cores next to lots of small cores creates accelerators for exactly the opposite: highly sequential sections of code in a mostly concurrent program. These sections of code contain, for instance, a lengthy critical path that cannot be broken up into concurrent sections. These sections of a program can then be run on one of the few larger cores. This spectrum is illustrated in figure 2.2.

### 2.5.2 Specialization

A third design spectrum when designing multicore systems is one of specialization. How much hardware is spent implementing specialized functions in hardware such as hardware support for cryptography, vector operations or multimedia operations? The benefit of specialized hardware is increased performance for programs that use the operations that are implemented in hardware, at a cost of either increased core size or lower performance for programs that do not contain these operations (e.g., a
Chapter 2. Related Work

Figure 2.2: Design spectrum for mixing core sizes on a chip.

Figure 2.3: Design spectrum for specializing cores on a chip

A hardware cryptography module makes the core larger, allowing fewer other cores on the chip. A core optimized for arithmetic operations by swapping cache space for functional units in the pipeline will perform badly when running programs with lots of memory accesses). This design spectrum is illustrated in figure 2.3.

Specialization only makes sense if it is known in advance that these operations are frequent enough to warrant this investment. Or in other words, if the software applications are known in advance. This is true for embedded systems, where the entire chip is optimized towards executing a set of applications as efficiently as possible. However, for general processing chips, the set of applications cannot be known in advance, and might even change in the future, in which case the processor has been designed with hardware features that are no longer used. Therefore, in this general use case, a multi-core chip best avoids specialization. At least, the programming model should avoid incorporating fixed specialized features to remain compatible with future designs.

Avoiding specialization makes sense from another point of view: fungibility. By ensuring that every core is the same, they become fungible, meaning that any core is as good as any other. This has advantages in handling hardware errors and resource allocation. It also allows software developers to better model the expected performance of a chip when writing their applications which in turn can enable real-time applications.
2.6 Programming Challenges

The current situation is one where chips keep getting more cores, but programming them is hard. The act of writing multi-threaded programs is also known more generally as parallel programming. Unfortunately, the practice of parallel programming never became mainstream \[58\] due to its inherent difficulties:

- Synchronization. As long as the threads in an application do not access a shared state, each thread can be easily executed in parallel. This is also called embarrassingly parallel. However, in typical applications, this is not the case and threads must synchronize with each other to ensure the shared state is accessed atomically. For a programmer to both find concurrency and define threads with the proper synchronization is a non-trivial task.

- Determinism. Although the order of execution of instructions in a single thread is well defined, the order of execution of instructions in different threads is, by definition, not. From point of view of the application, the scheduling of threads on the available processing resources is an implementation-dependent issue, and even the implementation may not be able to make guarantees with respect to this order; run-time conditions can result in a different scheduling every time the code is executed. Thus, the behavior of a multi-threaded application is harder to predict and requires additional thought to verify that it will always perform correctly.

- Debugging. As a result of the previous point, it is possible that an application that has a bug in it only shows that bug on some occasions, depending on the scheduling; the outcome of a program can differ depending on the scheduling of the threads if the synchronization between threads is not implemented correctly. Unable to deterministically reproduce bugs hinders the programmer’s ability to debug a program.

These issues have prevented the adoption of multithreaded programming in the past; it is only now that the frequency increase has slowed and performance increases from a single instruction stream has stagnated, the industry realizes that multi-threaded programming can not be avoided \[59\].

One of the popular methods for multi-threaded programming is thread libraries: the mechanisms required for creating multithreaded programs are defined in a library that can be used with a programming language that was not designed for multi-threaded programming. Popular examples of thread libraries are POSIX threads \[60, 61\], MPI \[62, 63\] and Intel TBB \[64\]. All three define constructs to create and terminate threads and synchronize between them. Thread libraries are generally the favored option for writing multi-threaded applications due to its widespread availability and ease of use. Programmers do not have to learn new a language in order to use threads. However, the downside of using thread libraries is two-fold: firstly, threads are generally ill-suited to be implemented as a library to extend a language that was not designed for multi-threading \[65\]. Secondly, due to the software nature of libraries, thread creation and synchronization is costly, generally in the order of 10,000 cycles \[60, 61\]. As a result of these overheads, thread libraries are unsuitable for
Chapter 2. Related Work

exploiting fine-grained concurrency that cannot be easily batched into larger chunks (i.e. irregular concurrency). The only viable alternative is to embed the notion of threads into the programming language and embed management of threads into the hardware in order to have cheap thread creation and synchronization.

Unfortunately, chip manufactures have commonly delegated thread management to operating systems and compilers; the hardware they create is according to their views and software can “figure out the rest”. They expect operating systems to efficiently manage the many threads defined by programs with the (few) hardware threads they are given. They also expect programming languages and compilers to remove all the difficulties of defining and managing parallelism such that programmers can easily write programs with plenty of parallelism, which will then be compiled onto their hardware platforms.

This train of thought stems mostly from a resistance to change; it is expensive and risky for the hardware industry to change drastically to a new computing paradigm. Delegating the experimentation to software is both easier and cheaper from their point of view. In fairness, there are recent exceptions to this: Intel’s Single-chip Cloud Computer [34] is an experimental hardware platform disclosed to a selection of groups, meant to evaluate what a change in the hardware computing paradigm could result in. Nevertheless, the changes in this chip when compared to a traditional multi-core chip are limited: the SCC consists of a few dozen legacy cores without cache coherency; instead, software is responsible for figuring out which data to make coherent with which cache.

2.7 Hardware Concurrency Management

If chips are designed with multiple cores and/or with hardware multithreading, then a scheduler must exist to schedule the available concurrency in a program or whole system onto the available hardware threads. Traditionally, these schedulers are software-based, where context-switching is done entirely in software and the hardware is unaware of the software-defined concurrency. As noted in the previous section, this situation is a result of decisions by hardware designers to push complexity to the software layer.

The trouble with this technique is that it only works if the concurrency that is managed is regular and coarse-grained (e.g., server connections, large loop iterations); the size of the tasks should be quite a bit larger than the relatively large overhead of managing the concurrency in software. However, for fine-grained or irregular concurrency (e.g., small loop iterations, function calls), the overhead of scheduling it to the available hardware threads quickly negates the benefit of distributing the concurrency.

In order to make it feasible to exploit more concurrency, concurrency management must happen in hardware instead of software so that its overhead can be dropped by several orders of magnitude. This principle is not new [10 66 41 67 68] but has traditionally not been accepted in industry because it was seen a limitation to move general concurrency management from software to hardware. However, since processors are implementing more and more hardware threads, the threshold of concurrency has to be lowered to fully utilize all hardware resources.
2.8 Summary

This chapter described the previous and current techniques used to increase the performance of processors via parallelism. It described SIMD and instruction-level parallelism and their shortcomings when it comes to scalability and energy-efficiency. Thread-level parallelism in the form of hardware multithreading offers the best chance at performance increase, but using them correctly in software has been troublesome since hardware designers tend to push the problems related to communication and synchronization to software.

Dataflow has also been shown to be an interesting approach, albeit one that cannot be efficiently implemented in its purest form due to practical constraints. A multitude of architectures have been designed that combined dataflow with the more traditional von Neumann architectures. Although these architectures look promising, each have their own pitfalls.

Taken together, we can conclude from this chapter that a scalable approach for the future lies in a multi-core processor with multi-threading to tolerate long latency operations whose parallelism should be managed in hardware as much as possible in order to allow for irregular, fine-grained concurrency to be exploited as well. This thesis, starting from chapter 4 describes the design and evaluation of exactly such an architecture.
Chapter 3

Microthreading

“Any sufficiently advanced technology is indistinguishable from magic.”
- Arthur C. Clarke

This thesis describes an architecture that was designed to efficiently run implementations of the Microthreading programming model. In order to understand the context in which the architecture was developed, and some of the decisions that have been made, the programming model that lies at its heart should be understood. Although this programming model was designed in parallel with the architecture to the point where changes in one affect the design of the other, its design is outside of the scope of this thesis. However, it should be pointed out that exactly because the research described in this thesis has been carried out in parallel with the design of the Microthreading model, it has also influenced and guided the model’s design. In particular, the main contribution of the work described in this thesis to the Microthreading model was the introduction of the concept of Window size (section 3.1.2) in the model as well as a re-definition of the existing concept of places (section 3.3) to be more generic and efficiently implementable in the architecture.

Microthreading evolved from previous architectural work called DRISC \[69\], a hybrid dataflow model. From DRISC’s specific architecture, Microthreading was developed as a generic programming model. The innovation in Microthreading’s programming model, compared to other parallel programming models, comes from not creating single threads, but instead creating ordered sets of threads called families and restricting the patterns of communication to allow for deadlock freedom and composibility of programs.

3.1 Families of threads

Programs in the Microthreading model are concurrent compositions of ordered sets of SPMD threads called families. Each family represents a unit of work (e.g., loop, function) with certain properties that describe how and where it should be run in the system.

A family is created by a single thread (called the parent thread), which is the only
Chapter 3. Microthreading

Figure 3.1: Concurrency tree in a microthreaded program. Every thread can create a family of threads of its own and barrier-synchronize on its completion. In this figure, thread A creates family B with $n$ threads. The threads in a family can communicate through a unidirectional communication channel (the dashed lines with arrows), effectively creating a dependency chain between them.

A thread able to synchronize on the family’s completion. Family completion is defined by every thread in the family having run and terminated, and all memory writes ensured to be consistent with at least the parent thread (see section 3.2.2). A family can have two kinds of named arguments: globals and shares.

Globals are read-only arguments that are provided by the parent thread and are available for reading in every thread of the family. Shares are named arguments which are shared between each neighbouring pair of threads in the family: reading a share will return the value written to the share with the same name by the previous thread in the family and conversely, writing a share will send the value to the shared with the same name in the next thread in the family. The border cases are handled by linking them back to the parent thread: reading a share in the first thread in a family returns the value written to it by the parent thread and writing to a shared in the last thread in a family sends the value back to the parent thread. These communication channels are illustrated in figure 3.1.

Both the globals and shares arguments have non-blocking writes and blocking reads. Combined with the special read/write semantics for shared arguments, this essentially turns them into a one-way chain for information. In contrast with the more general CSP model, this restriction on synchronized inter-thread communication allows for several benefits:

- Optimized virtualization. As will be described in chapter 4 by only allowing one-way communication between threads in a family, the hardware can more efficiently virtualize a family with a large number of threads onto cores with a limited number of thread contexts.

- Increased locality. Aside from the globals, which are a broadcast write-once, read-many argument, the main inter-thread communication is local. By ensur-
ing that neighbouring threads are placed closely to each other (i.e., on the same or a neighbouring core), communication via shareds is guaranteed to have low latency.

- Deadlock avoidance. The bane of communication in CSP is the possibility for two threads to both wait for a message from each other. Although such a problem can still exist in Microthreading on higher levels, the fundamentals of the programming model do not allow for it on the lowest levels.

Additionally, each thread in a family also has access to its unique index in the family. This value is the value for that thread in its family’s *index sequence*. This gives threads the flexibility to perform the same operation on different data (SPMD), or do different things altogether (MPMD).

Sequential analogies to the family are loops and function calls, which both capture regularity and locality in that model. All threads within a family are allowed to create families of their own, thus allowing microthreaded programs to capture concurrency at many levels by creating a hierarchy of families—a *concurrency tree*—as illustrated in figure 3.1.

This concurrency tree will evolve dynamically in an application and can capture concurrency at all levels, e.g. at the task level and, due to the threads’ blocking nature, even at the instruction level.

### 3.1.1 Index sequence

When a program creates a family, it specifies the number of threads to create via an integer *start*, *limit* and *step* value of the index sequence of the family. These indices work exactly as in a for-loop in C; a positive step defines one thread per iteration from *start* up to *limit* and a negative step defines one thread per iteration from *start* down to *limit*. In both cases, the limit is excluded and a limit that lies below or above the start index, respectively, causes no threads to be created. Following are some examples of traditional C loops and their start, limit, step equivalents.

<table>
<thead>
<tr>
<th>C loop</th>
<th>Start</th>
<th>Limit</th>
<th>Step</th>
</tr>
</thead>
<tbody>
<tr>
<td>for (i = 0; i &lt; 10; i++)</td>
<td>0</td>
<td>10</td>
<td>1</td>
</tr>
<tr>
<td>for (i = 14; i &gt; 3; i-=2)</td>
<td>14</td>
<td>3</td>
<td>-2</td>
</tr>
</tbody>
</table>

### 3.1.2 Window size

An issue with the expression of large loops as independent threads in hardware is the resources required to create them. This problem has been studied before[70] and the same solution of loop bounding has been adopted: to avoid rampant resource usage when creating families with a large number of threads, an allocation *window size* can be specified during family creation. This value specifies the upper limit for the number of threads created on a single execution core. By properly choosing a value during family creation, a program can allocate only a few resources to a family, leaving more resources available for other, perhaps more important families. Without specifying a window size, every execution core will attempt to fill up all its resources with threads from the family.
Chapter 3. Microthreading

3.1.3 Family synchronization

A thread can synchronize on a family of threads, meaning the thread suspends, if necessary, until the family has terminated. This means that all threads of the target family have terminated and any side-effects of those threads are made consistent across the system. Threads in a family terminate by executing a special kind of thread termination instruction. When a family has terminated, all of its dynamic synchronizing state is lost and all of the shared memory state (see section 3.2.2) that it has modified becomes defined.

3.1.4 Breaking a family

Any thread may break its own family (and that family only). When a break on a family has been issued by one of its threads, the creation of new threads ceases and all currently active threads are allowed to finish. This mechanism exists purely as a consequence of the window size mentioned before. The window size limits the number of simultaneously active threads and is equivalent to a pool of threads that are used to iterate over the family’s index sequence. The break mechanism allows families to be halted before its index sequence has been iterated over and is useful for efficiently implementing large loops with dynamic exit conditions as a family of threads with a relatively small window size.

3.1.5 Complex loops

Note that there is no hardware support for unbounded (infinite) loops (e.g. while (true)). Typically, these loops are ended by a (possibly) complex condition in the loop and the amount of parallelism is unknown, since it is not known in advance when the complex condition is met (if it was, the loop could be rewritten to a simpler form with an index sequence).

Although the number of iterations are not known in advance, such a loop can be implemented as a repeated creation of families of threads. In this case, the infinite loop is replaced by a new infinite sequential loop (i.e., implemented with branches) that creates a family of threads in each iteration and synchronizes on it. This family consists of some number of threads large enough to hide latency of its operations (e.g., 32).

Since most infinite loops of this nature require sequential semantics between its iterations, the family of threads can have a shared which each thread in the family reads at the start and writes at the end. If a thread matches the dynamic exit condition, it can break the family and write a special value to the shared to indicate that the following threads should do nothing.

Note that if the algorithm does not require sequential semantics (e.g., any match is acceptable, not necessarily the first), then the shared can be removed and the threads are completely independent.

1 The benefit of using threads over a loop is that the code can be optimized by putting all parallelizable operations (e.g., memory reads) before the read of the shared so they can be run in parallel.
3.2 Memory

Microthreading defines two kinds of memory: synchronizing and shared memory. The former provides synchronization on every read and write, but is small. The latter is large but only bulk synchronizes on entire families. Most programming models define only shared memory, but Microthreading chose to extend this with a separate, but small synchronizing memory to allow for fast inter-thread synchronization of small pieces of data that could be implemented efficiently. In fact, the global and shared arguments mentioned before are implemented using this synchronizing memory.

3.2.1 Synchronizing memory

The model specifies that every thread has a context of fast memory which is special in that it is a synchronizing memory. This memory provides the mechanism by which threads in a family can synchronize with each other, their creating thread and themselves. Each thread is allocated a context of synchronizing memory when it is created; when the thread terminates, its context is discarded. Each synchronizing memory location provides dataflow-like synchronization with blocking reads and non-blocking writes. Dependencies between operations in the same or different threads are enforced by this synchronizing memory: an operation on this memory cannot proceed unless its operands are defined, i.e. have been written as a result of executing other operations, including loads from shared memory.

Synchronization between threads in the same families is effected through this synchronizing memory by overlapping contexts for communicating threads, causing the sharing of synchronizing memory, creating a communication channel. However, a constraint is imposed on this mechanism, which arises from the combined requirements of locality, speed of operation and deadlock freedom. A thread may only have a dependency on values produced by one other thread: its predecessor in the family’s index sequence. This constraint ensures that dependencies between operations in a family of threads can be represented as an acyclic graph, which in turn ensures freedom from communication deadlock in the model. These acyclic dependency graphs are initialized by the creating thread, and values generated by the last thread are available to the creating thread after the family’s termination.

3.2.2 Shared memory

To store or pass large amounts of data, the microgrid offers a shared memory that persists over threads which can be read and written by all threads (see figure 3.2). This memory, however, is weakly consistent and is only defined using bulk synchronization on family creation and termination. At any other time, because the microgrid is a potentially asynchronous concurrent system, there can always be locations in the shared memory whose state cannot be unambiguously determined. Values written by threads in a family are guaranteed to be well defined only on termination of that family. Thus, arguments passed via shared memory must be defined before the family is created and the family’s results are only defined when the family signals termination.
3.3 Places

Microthreading defines the concept of places. This is an abstract collection of computational resources where families can be executed. When a thread creates a family, it can specify the place where to create that family. The act of creating a family on a place different from where the creating thread is running is called “delegation”.

The concept of places has several benefits. Primarily, it allows programs to avoid resource deadlock by creating units of work on different sets of resources. It also allows programs to explicitly define data locality by scheduling subsequent units of work that work on the same data on the same place. Furthermore, places can have certain properties, which can be requested by a program from the operating system when requesting places in order to execute some work at the best place. This allows microthreading implementations to run on both homogeneous as well as heterogeneous systems.

Some properties that might be defined by systems, are:

- Sequential performance. A program typically knows whether a certain unit of work has a lot of concurrency or not. This is usually an unavoidable result of whatever the code is trying to achieve. With places, the program can schedule the work at a place that is best suited for it.

- Instruction Sets. Instead of a program being written in a single instruction set, a program could exist of a collection of instruction sets, because the other instruction sets are significantly better suited for a particular situation.

- Memory bandwidth. With heterogeneous systems it is also possible to support a heterogeneous memory network. Certain areas of the system could have higher memory bandwidth to support memory-heavy work.

One particular important property should be highlighted. This is the property of mutual exclusion. If a place provides this property, then only one family will be
executed on a place at a time, with sequential inter-family memory ordering. This allows sections in a program, or even unrelated programs, to operate on shared data. If a particular implementation can make it very cheap to create exclusive places in a distributed manner, than this can be used as an implementation of traditional mutexes instead of e.g. an atomic test-and-set in shared memory. Using exclusive places instead of memory-based exclusion avoids pressure on the memory system, especially in distributed memory systems where the benefit is even greater as the “data” would get copied and invalidated for every processor that wants to lock it, and it preserves locality of the data that the exclusion protects.

3.4 Implementations

Microthreading is a programming model, and can have several implementations.

The simplest, and yet perhaps the most important implementation, is the sequential implementation. The inter-thread communication channels are defined such that Microthreading can be fully sequentialized. This is an important feature for several reasons. First, it allows a microthreaded program to be automatically compiled to an equivalent sequential program which can be run on contemporary hardware. This allows both the programmer to verify his code, and the hardware architecture to verify the output of the hardware (also see section 6.4). Secondly, the sequentialization of microthreaded programs is a crucial requirement for the microgrid architecture. The inter-thread state of a family of threads can be captured in a small, finite state. This allows the hardware to spawn as many threads as it can fit. If Microthreading did not offer this feature, any hardware implementation would have no choice but to instantiate every thread in a family at once.

Another significant implementation of Microthreading is POSIX threads (also known as pThreads). Microthreads can be trivially transformed into pThreads using C++ techniques for thread management and inter-thread communication semantics. With this transformation, every thread in a family becomes a pThread\(^2\). The performance will not be as great as the hardware implementation, but this allows a more detailed analysis of the concurrency aspects of a program. In addition, it allows interfacing with custom C++ libraries to perform more complex analysis on microthread programs \[^71\][^72]\[^73\].

Lastly, microthreading can be implemented in a hardware architecture. In this case, thread and family management and inter-thread communication are implemented by hardware and exposed to software via special instructions. This thesis concerns itself with this case and what such an architecture would look like, keeping efficiency and generality in mind. The resulting architecture is presented in the next chapter.

\(^2\)For efficiency reasons, these threads can be pooled to avoid creating hundreds of threads
3.5 Contrast with other programming models

The Microthreading programming model stands contrasted with other popular programming models:

- **POSIX threads.** POSIX threads is a POSIX standard for threads, published in 1995, which defines a library for C/C++ that offers an interface to the programmer which can used to create threads and synchronization primitives. It is a low-level library because thread creation and mapping is left to the programmer. It offers barebone access to threads and is intended to expose multithreading to a language that does not support it natively.

- **OpenMP.** OpenMP is “an API for multi-platform shared-memory parallel programming” [74]. Initially released in 1997, it is designed to be used by C/C++ or Fortran programmers and is used to identify blocks of code that can be run in parallel. OpenMP takes care of creating threads and distributing the threads across them.

- **Intel TBB.** The Intel Thread Building Blocks, much like POSIX, is a library with threading functionality. Intel TBB was released in 2006 and is more extensive than POSIX as it, unlike POSIX, offers a wide variety of data structures and primitives that enable more efficient and error-free multithreaded programming.

- **CUDA.** CUDA—or Compute Unified Device Architecture—is a parallel programming model developed by NVIDIA in 2007 with as primary target the Graphics Processing Units (GPUs) developed by NVIDIA, released in 2007. It consists of two parts: a C/C++ library as the primary interface to programs and a special C/C++ derived language that is compiled to native binaries to be run on the GPU. CUDA is positioned purely as an accelerator, since selected sections of code should be rewritten into the CUDA language and managed via the CUDA interface. The language is designed to run efficiently on a GPU architecture and does not scale to general purpose computing.

- **Cilk.** Cilk [25] is general purpose programming language designed for multi-threaded parallel computing. Cilk started in 1994 at MIT, but has only been targeting general purpose computing since 2008. It is owned by Intel Corporation since 2009, which has extended Cilk to be compatible with C++, under the name Cilk Plus. Cilk most closely resembles Microthreading in spirit by allowing the programmer to expose parallelism and having the environment take care of the scheduling the work over the available processors.

- **Go.** Go, or Golang, is a concurrent system programming language developed by Google in 2009. It is a garbage-collected C-based language with built-in support for thread creation and inter-thread channels.

Most of these programming models allow thread creation as single threads only. OpenMP allows bulk creation, but as a C/C++ API, it is still implemented using unit thread creation. The communications models in these programming models are also
more generic than Microthreading’s, allowing the programmer to more easily create deadlock situations.

3.6 Summary

In this chapter we described Microthreading, a programming model designed to be run on the architecture described in the next chapters. We described the concept of families: collections of threads with identical starting addresses but variable run-time code paths and how they can be used to compose a concurrency tree. We described the control primitives for creating a family, synchronizing on a family, aborting a family and sending and receiving data to and from a family. We described the difference between standard, shared memory and the fast, synchronizing memory and how the latter can be used to efficiently communicate data between threads in a family. Finally, we described the concept of places, an abstract collection of computational resources where families can be executed.

The model, although not a direct focus of this thesis, was co-designed along with the architecture and was, as such, influenced by us. Concepts that were introduced in this programming model as a result of the research in this thesis are the Window size as explained in section 3.1.2 and a re-definition of the existing concept of places as explained in section 3.3.
Chapter 4

The Core Architecture

“Perfection is finally attained not when there is nothing left to add, but when there is nothing left to take away.”
- Antoine de Saint-Exupéry

This chapter describes the main result of this thesis: the novel aspects of the design of a dataflow/von Neumann hybrid architecture with hardware multithreading that uses a large register file as the synchronization namespace for split-phase long-latency operations. The core architecture has been designed for composition into a larger microgrid which can be divided into program-specified partitions down to 1 core, where every partition is capable of executing either entire applications or parts of an application under varying resource constraints.

The architecture described in this chapter has, through several design cycles, been independently re-designed from the ground up to deal with various inefficiencies and bottlenecks that were present in the prior proposed Microthreaded architectures at the time this research started, as well as to implement new research insights that were introduced in parallel through research in programming languages, compilers, operating environments and memory interfaces targeting this architecture [76, 77, 78, 79, 80] (see section 4.14).

The architecture presented in this chapter has generated and supported a lot of the research mentioned above. Some of this work has been done in the same group as the core architecture design, and therefore our design has directly benefited from it; nevertheless, the core architecture is a comprehensive research topic in itself, despite requiring several layers of software to prove its feasibility.

This chapter describes the architectural components of a single core in this microgrid, whereas chapter 5 describes the chip architecture and chip-wide interfaces.

4.1 Fundamentals

The idea behind a dataflow and von Neumann hybrid architecture is not new. Iannucci argued in [81] that any architecture would have to deal with the latency of operations and the cost of synchronization. To solve these issues in a von Neumann
architecture requires a dataflow approach in the architecture itself with cheap and efficient synchronization. Particularly in the form of split-phase operations with I-structures for synchronization. This approach allows unrelated instructions to be executed between the issue of a long-latency operation and the use of its result.

Every core in the microgrid is exactly that: a dataflow/von Neumann hybrid processor that uses I-structures for the synchronization of split-phase operations. The I-structures were chosen to be implemented in the form of the core’s register file; all long-latency operations such as memory loads, floating operations and family synchronization occur via registers. Much like the HEP (see section 2.3.3) each core has multiple register contexts to support the multiple thread contexts, except in a microgrid usage of these contexts can be optimized in case of threads that do not require the full context, allowing the register file to have fewer registers than the case where it must provide a full register context for every thread context. Execution of the threads on a core occurs by interleaving them in a single pipeline based on compiler-provided information.

Modern multithreading processors such as the SPARC (see section 2.3.4) only support hardware thread switching, leaving software with the responsibility of filling the hardware thread contexts with created threads. As a result, overhead for thread creation and synchronization is relatively high, meaning that software does not expose all possible concurrency in a program as threads because these threads would be too small to be worth creating. Conversely, thread management in a microgrid is implemented in hardware, lowering the overhead of creating threads in bulk to the order of a dozen cycles. For functional concurrency—where only a single thread is created—this means that functions larger than a few dozens instructions are worth creating as a thread. For data-parallelism—where multiple homogenous threads are created—this means that threads are worth creating even when they are just a few instructions long, provided there are enough instances of them to warrant the overhead of creating them in bulk.

Another guiding principle for the architecture is maximizing energy efficiency by avoiding speculative operations such as branch prediction. Operations that could cause a pipeline flush are instead tagged by the compiler and cause a thread switch at the start of the pipeline to minimize the cost of a pipeline flush. To efficiently support this, the architecture performs thread switches with extremely low overhead; it is able to switch threads on every cycle. Note that this mechanism is different from interleaving as the thread switch tag can be ignored if no other threads are available, thus maximizing pipeline usage in situations where there are fewer threads. Because thread switches are extremely cheap, compilers can be oblivious to the details of scheduling and non-unit instruction latencies and tag the instruction stream based on its static properties.

The architecture has also been designed to allow for scalability of up to hundreds of cores on a chip. By avoiding large caches and providing latency tolerance, the core size can be kept reasonably small despite the large register file (see section 4.13). This allows up to hundreds of these cores to be fitted on a chip. In such a case, the dynamics of multiple multithreaded programs running concurrently, changes. Rather than time-sharing a limited set of cores with a limited set of threads, the chip would be space-shared, with multiple programs running in parallel on different sections of
the chip and each section time-sharing between threads of a program or unit of work. As such, the architecture has to support situations where programs send work off to far-away areas of the chip for a relatively long time.

Finally, in order to assist commercial adoption, backward binary compatibility with an existing instruction set should be guaranteed for most if not all application code. Fortunately, this is possible because the instruction set is extended with thread and family instructions. In fact, given that the split-phase mechanism for long latency operations is entirely transparent to the instruction set, the architecture can even execute legacy code from the unextended instruction set while getting the general benefits of the latency-hiding and hardware multithreading. Furthermore, by ensuring that the instruction set does not require knowledge of architecture-specific parameters, application binary code can be both backward and forward compatible across older and newer generations of the architecture with different configurations.

The rest of this chapter describes the architecture that has been developed with these principles in mind.

4.2 Overview

A core in the microgrid is a classic von Neumann core with several extensions to support hardware multithreading. Figure 4.1 shows the components on a microgrid core and their signaling inter-dependencies. Familiar components include the pipeline, I-cache and D-cache, register file and interface to a mesh network. The added components are a thread scheduler, thread and family allocator and the thread and family table. There are added data paths to connect the new components and the familiar components have been extended to support thread and family management.

The rest of this chapter and chapter 5 describe how the components on a core interact with each other and other cores in order to implement the execution model outlined in the beginning of this chapter.

4.3 Thread & Family Table

The thread table and family table are the data structures that contain all hardware thread and family contexts on a single core. The family table contains, among other fields, the program counter for new threads, the next index value for new threads, the number of threads left to create and information to place this family context in a multi-core family. The thread table contains, among other fields, the current program counter of the thread and offsets into the register file for the thread’s register context. In short, the family table contains all thread-invariant information that is family-specific, whereas the thread table contains all thread-specific information. The exact contents of these tables can be found in appendix A and B.

Note that although they are named tables and each presented as a single data structure, they do not physically have to be implemented as such. In practice, both

1For instance, the latches in the pipeline have been extended with a thread ID to identify the thread that the instruction belongs to
Figure 4.1: High level block diagram of the components on a core. Shown are the standard components (I-Cache, D-Cache, Pipeline, Register File and Network) as well the additional components which implement the thread management, and the memories which store the thread state.
the family and thread table can be broken up into multiple sub-tables based on the usage of their fields. Each sub-table can then be placed near the components on a core that use its information, minimizing wire delays and layout complexity. Similarly, fields in the sub-tables can have different number and types of ports depending on their use.

The size of these two tables directly correlates with the amount of latency a single microgrid core can tolerate. With a larger thread table, more hardware threads can be active, and longer periods of long-latency operations can be tolerated by executing these threads. However, because each thread also requires a register context, a larger thread table typically necessitates a larger register file (see section 4.4), which is one of the largest components on the core (see section 4.13).

The size of the family table can be set relative to the size of the thread table based on the type of core. A core that is mainly used for data-parallel operations can have a family table that has a fraction of the entries of the thread table, or even just one entry. Conversely, a core that will run mainly functional concurrency will run many families of a single thread, requiring a family table of equal size as the thread table. It would even be possible to create a heterogeneous microgrid with different sizes for the thread and family table on different cores, allowing certain cores to specialize towards functional concurrency, and others towards data concurrency.

### 4.4 Register file

The register file in a Microgrid core acts as the synchronization namespace for dataflow operations. As illustrated in figure 4.2, registers are extended with a state bit that indicates whether the register is full or empty. In the latter state, the register’s value bits contain continuation information in the form of a head and tail pointer to a linked list of thread contexts that should be woken up when the register becomes full (see also section 4.9).

Because all of the instruction set’s general purpose registers are mapped into this register file, all general purpose registers have this synchronizing ability. The synchronization happens implicitly, without changes to the instruction set, and on all instructions; any instruction that reads from a register that is not full will cause the thread to suspend and any instruction that writes to a register sets its state to full.
Long-latency operations such as memory loads are transparently implemented as split-phase operations; executing the instruction issues the operation and sets the destination register to empty. When the operation has completed (e.g., the data has been fetched from memory), the register is written and set to full, waking up any threads that were suspended on it.

Although this behavior requires additional logic throughout the entire pipeline to support these states from the register file and bypass buses, the benefits are the ability to have dataflow synchronization on every operation in the pipeline, backward compatibility due to an unmodified base instruction set and the combination of both; for instance, memory loads in legacy binaries have these latency hiding benefits.

Besides supporting split-phase long-latency operations, the registers in the register file are also used for inter-thread communication. As described in chapter 3, communication between threads and between families of threads occurs via so-called global and shared channels. The former is a communication channel from the parent thread to every thread in the child family. The latter is a uni-directional chain of communication channels between the threads in a family. In this architecture, these communication channels are also implemented using the register file and its synchronization features.

For the following discussion, the distinction between a thread’s logical and physical registers must first be defined. The former are the thread’s registers from the point of view of the instruction set. For example, in a typical RISC architecture, this can be $r0$–$r31$. The latter defines the registers as they are in the register file, which can be $r0$–$r511$, in the case of register file with 512 registers.

### 4.4.1 Logical context

Threads running on the microgrid have their logical register contexts split up into four regions:

- **Locals.** These registers are private to the thread.

- **Globals.** These read-only registers are shared by all threads in the family and are meant for loop invariants such as pointers or pre-computed constants. Their contents is provided by the thread that created the family.

- **Shareds.** These registers can be read by the next thread in the family via its “dependent” registers. Combined, these two register classes create a private, one-way communication channel between two adjacent threads in a family. Since such a channel exists between every adjacent pair of threads in the family, this creates a uni-directional fast-access and fast-synchronization communication channel for small (register-sized) data.

- **Dependents.** These registers contain the value written to the “shared” registers of the previous thread in the family.

\[^2\text{As long as independent instructions are scheduled between the memory load and the use of its target register.}\]
Figure 4.3: Logical context layout of a thread’s integer registers with 8 locals, 4 globals and 6 shareds and dependents. On the left are the base ISA’s register specifiers. On the right are their alternative specifiers which indicate the sections. Note that only 24 of the 32 registers are used; $r24$–$r31$ are read-as-zero. In this example the base ISA is from the DEC Alpha.
Chapter 4. The Core Architecture

The compiler determines the size of these four regions (with the shareds and dependents regions of equal size) based on the thread code, with the constraint that all these sections must fit in the instruction set’s register context. An example mapping of these regions into the logical register context for a RISC architecture with 5-bit register specifier is shown in figure 4.3. Note that any or all of these sections can be of zero size.

4.4.2 Physical context

The regions from the logical context as described in the previous section are mapped onto the core’s physical register file. The thread table contains pointers into the register file, identifying the thread’s locals, shareds and dependents. The pointer to the globals region, as well as the size of each region, is stored in the family table, as this is shared by all threads. The pipeline translates the 5-bit register specifier from the instruction into an address into the register file based on this information. An example of this mapping is shown in figure 4.4. Note that the shareds of thread $i$ physically overlap with the dependents of thread $i+1$. By mapping these two regions onto the same registers, the synchronizing properties of the registers can be used to create the efficient communication channel. When a thread is created, its shared registers are set to empty such that thread $i+1$ can read any one of its dependent

---

3 The 5 bits come from the base ISA that has been chosen in this example.
and suspend on it until it is written by thread $i$. In fact, if the instruction that reads the dependent follows the instruction that writes the shared in the pipeline, the value can be picked up from the bypass bus, as bypasses are based on the register’s physical address.

There are a few interesting points to note in figure 4.4. First, the dependents of the first thread in the family have no shares to be mapped onto, so they have their own section of physical registers. The thread that created the family can fill these registers with a special inter-context register move instruction. Similarly, the shares of last thread in the family have no dependents mapped onto them. Instead, the thread that created the family uses another inter-context register-move instruction to copy their contents back to its own context.

Secondly, the shares of threads 2 overlap with the dependents of thread 0. At first glance, this might cause writes to thread 2’s shares to overwrite the dependents of thread 0. However, because the allocation window size (see next section) is 2, at most two threads will exist at any time. Thus, if thread 2 exists, thread 0 must have terminated, freeing its locals and dependents for reuse.

4.4.3 Allocation window size

Because the execution model consists of hierarchies of threads, a tree of multiple families of threads can quickly grow and occupy all hardware contexts, subsequently requiring additional contexts before it can finish, thus causing a resource deadlock situation.

One solution to resource deadlock is to virtualize the resources; the hardware, operating system, or both would write used contexts to memory, making them available for reuse by the hardware. When a swapped-out context is required, a hardware trap would fire on its access and cause the required context to be swapped with another context. This solution has not been adopted because it would significantly increase the complexity of the hardware, whereas it is not even clear that in a multithreaded environment, contexts will remain unused for a long enough period of time to warrant them being swapped out to memory (or in other words, that thrashing would not occur).

Instead, the simpler method of controlling how many threads of a family should exist at any given time has been adopted. This amount of threads, called the allocation window size, can be specified by software when a family is created. This window size specifies an upper limit on the number of threads allocated at any time to this family, per core. By controlling this value, the compiler or runtime environment can constrain the width of the parallelism to allow it to run deeper given the available resources.

For instance, for families at the leaves of the concurrency tree that perform a lot of independent work, a large window can be used to create as many threads as possible, whereas higher up in the tree it might be better to specify a small window to keep thread contexts available for later creates.

Note that in cases where the depth and/or width of recursion is not known in advance, it is not normally possible to choose a window size such that resources do

\[\text{4A special instruction is required, because the parent thread would have no regular way to address the child context’s registers.}\]
not run out and deadlock does not occur. This issue has been solved in the Microgrid via *automatic sequentialization*, which is explained in section 4.7.

### 4.5 Family creation

A family of threads is created in three steps, where each step is implemented with one or more instructions. First, software specifies a set of cores that the family will run on (also known as a *place*, as described in section 3.3) and a family table entry is allocated on each core. Secondly, software sets up the allocated family entries with the properties of the new family. Finally, software “creates” the family, indicating to the hardware that the setup is complete and that it can start creating and executing threads for the family.

This mechanism was favored over the alternative of having a single instruction with a memory pointer to a data structure that contains all the information, for two reasons. First, with nested creates, any code that creates a family can be executed many times in parallel. If the properties of the to-be created family are not static and multiple threads wanted to create the family, an instance of the data structure would have to be created on each thread’s stack, using memory space and bandwidth. By using instructions instead, such dynamic properties can be calculated in the thread and stored in its registers instead. Secondly, certain values for properties are common; for instance, it is feasible that a start index of 0 or a step size of 1 are common properties for a new family. By using instructions and initializing all properties with a default value when the family table entry is allocated, some instructions may be omitted. This optimization cannot be used if all family properties are stored in a data structure.

#### 4.5.1 Dependent families

Families that have one or more shareds (see section 3.1) are called *dependent* families, as every thread in that family may be dependent on its predecessor. Opposed to these kind of families are *independent* families, where every thread can execute independently of its siblings. The dependency in dependent families does not come from some intrinsic property of the family that is set when it is created, but rather from the dataflow semantics of the variables that are shared between consecutive pairs of threads; a family that defines shareds but does not use them can essentially run as an independent family.

However, because of this (possible) dependency, hardware must treat such families differently. As explained in section 4.4, the dependents of a thread are registers that physically map onto the corresponding shared registers in its predecessor. This means that a thread’s register context *cannot be cleaned up or reused until its successor has terminated*. This rule trivially stems from the register mapping between them. Since the hardware cannot know when a thread’s successor has finished using its shareds, it can only assume it has finished using them when it has terminated. This requirement necessitates hardware to enforce the rule by setting a flag in a dependent family’s
thread’s predecessor context whenever it terminates\(^5\).

When a family is created on a single core, the thread allocation process is atomic and serial by design. When a thread is created, its predecessor always already exists due to this nature of the thread allocation process on a core. However, when creating a dependent family on multiple cores, two problems exist. First, different threads in the family can be created at the same time. Inter-core communication would be required to manage both the dependencies between threads as well as exchange the shared registers. However, when a thread terminates, it is possible that its predecessor on the previous core has not yet been created. In such a case, the thread termination event must be buffered in the family entry and care must be taken such that no such second event can occur before the first one has been handled. This mechanism requires non-trivial logic and is non-trivial to verify for correctness.

The second issue is one of memory consistency. With an independent family, the threads in a family do not need to communicate with each other, so the memory consistency model where a thread’s writes are only visible to the parent after the family synchronizes, suits it well (see section \([5.4]\)). However, with a dependent family, there is a communication channel between threads. This channel can be used to signal to the next thread that a region in memory has been updated by the current thread. For instance, a dependent family can be created to create a histogram; in order to ensure that each thread updates the histogram atomically, it can be serialized by reading and writing the shared variable before and after the update, respectively. In such a situation, the next thread should be guaranteed to have access to the updated memory after reading its shared variable. To provide proper consistency semantics in this case, the memory protocol would need to conservatively order all stores between the cores involved with the dependent family, regardless of whether this synchronization would be needed at all. Such an implementation would come at a hefty price which is not well justified by the meager performance increase made possible by parallelizing dependent families on multiple cores.

Therefore, to avoid these two issues, dependent families are restricted to single-core creates only\(^6\). Although this might seem to result in a severe restriction of the parallelism of the family, this is mitigated by the fact that a dependent family by its definition runs largely sequentially\(^7\). A thread in a dependent family is typically constructed of a prefix, main body and postfix. The first dependent and last shared are read and written at the start and end of the body, respectively. The prefix and postfix are independent. Since, in general, the body of such a thread cannot run until the body of its predecessor has finished, this makes the prefix and postfix the only scalable source of parallelism. In cases where this parallelism is significant, such a family can be split up into two or three separate families consisting of the prefix, body and postfix, respectively, where any intermediate state can be stored in memory (see \([5.4]\)).

\(^5\)Conceptually, it would be possible to have the compiler emit an instruction at the point where it knows a thread will no longer use its dependents, but doing it implicitly at thread termination saves an instruction and avoided the additional work needed to get a working compiler.

\(^6\)An analysis of how and when parallelizing dependent families is beneficial in the first place can be found in appendix \([C]\).

\(^7\)Despite the inherent sequentialism of dependent families, they still have an advantage over a for loop: the sections of codes that are independent will be executed independently and can be used to hide latency.
The independent prefix and post families then benefit from being able to be distributed across multiple cores. Since the cost of creating a family is relatively low (as is explained in the rest of section 4.5), this solution has very little overhead compared to having a single family.

4.5.2 Register count word

As described in section 4.4, a thread’s register context is divided into globals, shareds, dependents and locals. Since this division can differ between families, this information has to be passed to the hardware on a create. However, unlike the other properties of the family such as the number of threads and its place, this property is not determined by the thread creating the family. Instead, it depends entirely on the code of the created family. The compiler passes this information to the hardware by placing a special pseudo-instruction called the register word immediately before the first instruction of the thread, which is subsequently read by hardware when creating a family.

To avoid the logic required for hardware to read in a cache-line for this register information, other possible solutions revolved on the register information being set through an instruction instead. However, the information still had to come from the “callee” somehow. In the worst case, the “caller” does not know this information about the callee at compiler time. This happens, for instance, when the callee is in another module (e.g., library). Although this information could have been made available at runtime through special linker or compiler extensions (e.g., defining special symbols), this would complicate existing tools and compilation methods. Instead, the more compatible method described above has been chosen at the cost of some hardware. This hardware method also has the advantage that the first cache-line of the created family can now be loaded through the I-Cache instead of the D-Cache as with other methods. This avoids a cache miss on this line by the thread scheduler when the threads are created (see section 4.9).

Allowing the compiler to specify the layout of the register file can present difficulties for compiler design. For a comprehensive explanation of these issues and how this architectural feature can be used in software in general, see [77].

4.5.3 Detached creates

Traditional family creates discussed up to now have always been part of a create/sync pair; a thread creates a family and later synchronizes on its completion. There are, however, instances where a thread may not want to synchronize on a family that it created. These sync-less creates are known as detached creates. This type of create is required to support common system patterns such as callbacks. Unfortunately, in order to support this type of create, certain restrictions are placed on the hardware.

For instance, after a family has been created on a set of cores, the global and shared registers from the parent thread must be transferred to that family. However, not all solutions for this issue can support detached creates. In one solution [82], these arguments are stored in the parent’s context and whenever the created family reads a global or shared register for the first time, a request would be sent that copies the
Figure 4.5: Example of splitting a dependent family of 8 threads with a large independent prefix (A) and postfix (C) into three families. Shown is the scheduling of the thread parts onto cores with 4 thread contexts in the case of splitting (right) and not splitting (left). By splitting the family, the entire computation can be completed faster because the independent prefix and postfix can be spread out across multiple cores whereas the dependent part (B) cannot. Because each thread is now split across three registers contexts, variables that are used in more than one thread part must be passed via memory, the cost of which is not shown in this example.
value from the parent’s context. However, with detached creates the parent thread can terminate before the created family does. In such an event, the register context would have to be kept around such that the created family can copy the registers from it. To guarantee proper cleanup semantics, complicated logic and administration would be required.

Instead, we recommend that the created family, after creation, is fully independent of the parent thread. This means that all global and first shared registers have to be explicitly pushed by the parent thread to the created family. Although this could have been automated in hardware, it would have required iterative logic and notification of completion for the parent thread. By following the mantra of keeping the hardware simple whenever possible, these register pushes have been implemented as instructions, leaving the compiler with the explicit responsibility of ensuring that it emits an instruction for every global and shared register that should be transmitted to the created family. Similarly, “pull” instructions were introduced to pull the last shared of the family back to the parent context after it has synchronized on the family. This solution is the one evaluated in this thesis.

This solution has the additional benefit of simplifying compiler development. With the alternative method of copying the parent registers when the created family uses them, the registers must be available for copying at any time between the create and sync. This means that the compiler cannot use those registers for anything, including spilling. It also means that the compiler’s working set of registers is reduced between the create and sync. If several creates are lexically nested, it would be possible to run out of registers with just a few nesting levels, depending on the number of globals and shareds in each family. By explicitly pushing and pulling the global and shared registers, no registers have to be reserved between the create and sync.

Note that besides supporting detached create, having instructions for pushing registers to families on other cores allows for arbitrary signaling between threads where registers are used at synchronizer objects. Exploiting this mechanism for system and software design is a topic for future research.

4.5.4 Parallel creates

Since the microgrid is a parallel system, family creates also happen in parallel. Because single contexts are allocated atomically in hardware, single-core creates are trivial to implement: they either succeed or fail because no more contexts are available. However, multi-core allocates can cause deadlock depending on the implementation. If a context can be allocated on some cores but not on others, the system could deadlock if the hardware suspends the allocate until those contexts are freed up: all contexts may be used by a program that requires the pending create to finish before it can release any contexts.

To solve this, the hardware was designed to atomically allocate a context on every core in the place. A token-based system, where only the core that has the token can send creates, was found to be too restricting as it serializes all multi-core creates. Carving up the cores on the grid into fixed-size places, each with their own token

---

8Obviously, this does not apply to detached creates
Figure 4.6: Multi-core allocation process. The process starts with the allocation request arriving from the mesh network at the first core in the place (1). The link network (see section 5.3) is used to allocate a context on every following core, one-by-one, until all contexts have been allocated, or allocation fails (2–4). In both cases, a message is sent back committing the contexts or releasing the contexts until a power of two number of contexts remain (5–7). The last step (8) returns a message over the mesh network to the issuing thread with the family identifier of the newly allocated multi-core family.

was found to be inflexible and it did not solve the issue of over-restricting multi-core creates for the larger sets.

Instead, family allocation was implemented as a try-to-allocate mechanism. When a multi-core allocation arrives at the first core in the place, a single context on that core is allocated (if this fails, the allocate fails as with a single-core allocate). Afterwards, the hardware will attempt to extend the allocation with a core at a time until a context has been allocated on all specified cores, or until a core is reached that has no available contexts. This process is illustrated in figure 4.6.

The allocation semantics include several flags that allow software to override this behavior and require all cores to be allocated. This can be used, for instance, if the actual number of allocated cores is important to the algorithm for mapping or alignment reasons and software can guarantee—for instance, through static analysis by the compiler—that no resource deadlock will occur.

4.5.5 Exclusive creates

In order to support exclusive places (see section 3.3), each core reserves one family table entry for so-called exclusive creates. These creates are marked as exclusive by the compiler, possibly via a separate instruction. Exclusive creates delegated to the same core have the guarantee that only one of them will execute at a time. Additionally, the memory model guarantees consistency between subsequent exclusive families.

Technically, since family creation at a high-level is split into separate steps for family allocation, family setup and family creation, it is really the family allocation that is exclusive. The exclusion is implemented as follows: each core contains a FIFO queue of incoming exclusive family allocation requests, either from the same core or delegated allocations from other cores. Once the exclusive family table entry is free,
the core will take the next allocation request from this queue and assign the exclusive family table entry to it. From here on out, the process is identical to a non-exclusive family allocation.

This extension to the family allocation process is trivial, yet opens up a powerful feature to software: the ability to have as many cheap exclusive places as there are cores. For details on how to use this feature from the software’s perspective, refer to [78].

4.6 Family termination

In any system, a piece of computation must, at some point, terminate. There are three methods to effect this termination: implicit, cooperative and pre-emptive termination. Implicit termination occurs when all threads in a family reach the end of their execution and terminate. Cooperative termination is any termination that occurs before implicit termination would occur and is triggered from the code itself, for instance, when the termination condition for the computation is not known in advance. Pre-emptive termination is a feature that is required in multi-user or multi-tasking system, as part of the security system. The reasons for pre-emptively terminating a computation can range from misbehaving code to hangs or timeouts or simply because the user requested it.

Since family and thread creation happens in hardware, family termination must also happen in hardware. The hardware supports non-implicit family termination through the “break” and “kill” operations. The former takes no arguments and implements cooperative termination by halting thread creation in the family that executed the instruction, allowing the existing threads to end gracefully.

The latter implements pre-emptive family termination; at first thought, the argument to the “kill” operation might be a family identifier, such that the operation kills that family. However, considering that a family can spawn families of its own, killing a single family is not enough; the entire concurrency tree rooted by that family must be killed. Iterating through the spawned concurrency tree in order to catch up with the leafs of computation takes considerable logic and in the worst case does not even guarantee that it will ever complete. Instead, a scheme was proposed where “kill” terminates the families on an entire core or set of cores. It is meant to be used in conjunction with the operating system, which should maintain a list of cores allocated to (for instance) every process. A process can then be terminated by the operating system, possibly on behalf of another application or the user. The operating system will issue the “kill” instruction for every core that has been allocated to the application while simultaneously disallowing further core allocations by the application. This will guarantee that the application will be terminated and cleaned up with only a per-core kill needing to be supported in hardware. This interaction with the operating system is described in more detail by van Tol[78].

In all of the three cases of family termination, the termination event is a synchronizing event that the parent thread might be interested in. The parent thread registers its desire to be notified of family termination via a sync operation, specifying both the family it wishes to be notified for, and a register that will be written once
the family terminates. The hardware will write this register when all created threads in the family have terminated (this covers both implicit and cooperative termination) and when all pending operations such as memory writes have been made consistent according to the memory consistency model.

4.6.1 References

A core might have references outside the core to its internal state, or have references from the core to the internal state of other cores. Because of these references, additional attention must be given to preemptive family termination via a core kill, as such an event must not result in an invalid internal system state.

References to the core

References to a core’s internal state may exist outside the core in the form of pending FPU operations, allocations, creates and syncs.

Specifically, pending memory loads are not a problem. When a memory load is issued, a reference of the destination register is kept in the L1 D-Cache (see section 4.11). Since the relevant lines in the D-Cache would be reset upon a kill, this is not a problem. The request that has been sent to the L2 cache returns as a broadcast operation on the bus connecting the L1 caches of several cores (see section 5.4). If the D-Cache has been emptied between the issue of the request and the arrival of the data, the broadcast is simply ignored on that core (i.e., the memory load might have been futile, but the internal state of the system is still valid).

All of the other references listed above contain a register address into the register file of the core that is used to write back some value once a long-latency operation completes. If the core is reset, these references become invalid. If these references are eventually used in a writeback operation, the register which they reference may have been allocated to a completely different family that has been allocated since the core was reset.

Unfortunately, this problem cannot be solved by waiting for these references to complete. Pending synchronizations may not complete for a long time, or never, if a resource deadlock situation occurs. Instead, one way to solve this problem is to store a random or incremental check value in each register as the long-latency operation is dispatched. This value is sent along with the operation as part of the remote register reference. When the writeback occurs, the check value in the reference is matched with the check value in the register. If they do not match, the write is ignored. The size of this check value has to be large enough to avoid duplicate values during the lifetime of outstanding references. How large this specifically is, depends on the system implementation. It’s possible that a system will choose to kill the places that the killed code delegated to as well. Aside from this issue, the impact of this solution is the increased bandwidth usage for remote writes caused by adding the check value to the message.
References from the core

Similarly, a core might contain references to the internal state on other cores in the form of pending allocations, creates and syncs. These operations all send a remote register write to a register on the core that issued the operation. As such, the other core will most likely be waiting for these writebacks before it can continue and terminate. If all activity on the core is killed without dealing with these references, parts of the system may deadlock.

Solving this problem requires cooperation with the operating system. The operating system is assumed to manage a list of atomic killable units of computation and administration, called components (e.g., what is nowadays called a process). Next to this list, the operating system is also assumed to have a list of resources allocated to each component. Creating new components requires passing through the operating system such that there are no direct references between components, but only to their handles in the operating system. Only components as a whole can be pre-emptively terminated by other parts of the system. The operating system is responsible for effecting communication between components. In such a case, when a component is killed, no other component can be left with family references to or from other components. The operating system can then ensure that these references are cleaned up appropriately.

4.7 Deadlock freedom

Any architecture must be able to guarantee freedom from deadlocks under common use. Three classes of deadlocks can be identified: communication deadlock, resource deadlock and hardware deadlock. Although these classes of deadlocks can fundamentally be considered the same, they have been separated here based on the responsibility for avoiding them: the program, the hardware or the system designer, respectively.

4.7.1 Communication deadlock

Communication deadlock is a deadlock caused by a communication pattern in the program. It is not a fault of a hardware, but of a logic error in the program. Although the execution model aims to sanitize the communication model between threads to avoid these deadlocks from occurring, this is no guarantee that communication deadlock will never occur. However, as this form of deadlock is essentially a software issue, the architecture need not take special measures to avoid it, but only needs the basic support needed to clean up and reset the hardware whenever deemed necessary by the operating system.

4.7.2 Resource deadlock

Resource deadlock is a situation where a program can no longer continue because it requires additional contexts (resources). However, all contexts that are in use depend on the program acquiring new resources in order to continue. Thus, the program deadlocks. This type of deadlock can occur because the program or programmer typically
does not or can not know the available resources of the architecture. This problem is worsened if different configurations of the architecture are available, each with different amounts of resources. The problem usually shows itself when the amount of resources used by a program depends on run-time conditions. In a multi-user or multi-tasking system, any number of applications can be running on the system, all competing for the same resources. Furthermore, the operating system may also constrain applications to use a quota of resources.

One familiar solution to resource deadlock is virtualization, where contexts (thread entries, family entries and registers) are swapped out to memory\textsuperscript{10}. However, this would introduce significant logic and latency to deal with the swapping of resources to and from memory, especially considering that in this architecture registers are synchronizing I-structures with thread continuations. As such, virtualization would compromise the simplicity and low-latency nature of the hardware.

Instead, another solution to resource deadlock has been chosen in the form of automatic sequentialization. When a program issues a create, the create can fail if it cannot allocate the desired resources. The program can inspect the return value of the create (the family identifier), see that the create failed, and execute a sequential fall-back path. This fall-back path uses the fact that families of threads can be executed sequentially as a for-loop; a property which is derived from the fact that communication between threads is unidirectional. If CSP-style communication was allowed between threads in a family, providing a automated sequential version of a family would be significantly harder, if not impossible.

The downside of this technique is the code duplication required to provide the sequential fallback path. The entire thread being created must be compiled twice; once assuming it is run as a thread and once assuming it is run as a sequential version. Special care must be taken in the programming language in order to support this, especially with function pointers.

This downside can be mitigated by allowing for the create to suspend in a hardware queue rather than fail if resources are not available. Although using this feature removes the guarantee of freedom of resource deadlocks, programs can use this variant if they are sure (for instance, through static analysis) that resource deadlock will not occur and the benefit of spreading out the work is worth waiting until resources are available.

\section*{4.7.3 Hardware deadlock}

The last form of deadlock, hardware deadlock, is caused by design errors in the hardware that allow a situation to occur where no hardware process can proceed. Cases of hardware deadlock can be caused by wrong arbitration on shared ports to data structures, insufficient buffer space to accommodate all messages or requests, and so on. All these issues either lead to the pipeline stalling indefinitely, or pending requests never completing such that threads wait indefinitely for a response.

Hardware deadlocks must be diagnosed and fixed during design of the architecture. This process, however, is complicated by the fact that, as with any parallel system,\footnote{Of course, the system can run out of memory, but memory is generally considered “large enough”.}
some deadlock situations only arise in specific circumstances. Section 6.5 talks more about how deadlocks were diagnosed and solved during the design of the hardware.

4.8 TLS

The thread creation and management system described in this chapter so far is capable of executing arbitrary threads. However, the number of registers available in a thread may not be sufficient. Firstly, a thread may have a working set of live variables that is larger than the maximum register context size. Secondly, as described in section 4.7 in order to avoid resource deadlock, a single thread must be able to run sequential code with functional calls. This requires stack space for storing the function frames as function calls are made.

Both of these conditions require that each thread should be able to have a section of private memory, also known as thread local storage. In contemporary systems, TLS is explicitly allocated by the operating system whenever a thread is created. This is feasible because thread creation is relatively infrequent. However, in an architecture where threads are created as often as functional calls are made in contemporary systems, explicit memory allocation for each thread is infeasible, as it would increase the latency of thread creation from mere cycles to hundred if not thousands of cycles.

The only alternative is that the hardware is at least partly involved in the process of allocating memory when threads are created. Since dynamic allocation on thread creation—whether in hardware, or through a software callback—is impractical, static allocation must be used; threads are statically mapped onto an address space and provided with a pointer to the top and bottom of their private memory. The question that remains is how much space to allot each thread, or in other words, what address range to divide among the threads in the system.

An approach was tried where a fixed portion of the address space (the top half) was divided among all threads on the system. On a 64-bit architecture with 1024 cores and 256 thread entries per core, this would give each thread 32 TB of memory; more than enough, even by tomorrow’s standards. However, it also means that if a thread does not require use of its TLS, 32 TB of memory have been needlessly reserved for it; a tremendous amount, even by tomorrow’s standards.

Instead, an approach has been adopted which lies in-between these two extremes of per-thread dynamic allocation and fully static allocation; TLS is dynamically allocated per core allocation. Whenever a set of cores is allocated, this operation must go through the operating system in order to set up the cores with the security capability. This operation can be paired with the allocation of a block of memory. A pointer to the top and bottom of the block of memory is set up on each core along with the security capability. Each core can then use these pointers to define the address range which is statically divided between the threads on the cores. This approach has the benefits of both methods; thread creation remains a low-latency operation, yet the application can indicate how much TLS is required when a chunk of work is distributed to a set of cores.

For a comprehensive explanation about this mechanism and its impact on com-
Chapter 4. The Core Architecture

4.9 Thread scheduler

The thread scheduler is the part of the core that ensures that threads that were suspended will now be run in the pipeline. To this end, it maintains a list of threads called the *active list* that the pipeline can pop a thread off whenever it performs a thread switch. This list, however, does not simply contain all threads that have woken up. Namely, if a thread’s line in the instruction cache has been evicted since it last ran, the pipeline would stall waiting for the cache-line to be fetched whenever it switches to that thread. To avoid these stalls, thread scheduling occurs in two phases.

The first phase is a list of threads called the *ready list*. All threads that can be run by the pipeline are placed on this list. The second phase is a process that pops a thread off the ready list and checks the I-cache. If the check results in a hit the thread is placed on the active list. If the check results in a miss, the cache-line is fetched and the thread identifier is stored in that cache-line in a so-called *waiting list*. If the check is a hit, but the data is still being loaded from memory, the thread identifier is appended to this waiting list. When a cache-line returns from memory, the threads on the returned cache-line’s waiting list are appended to the active list. By appending threads to per-cache-line waiting lists, instead of stalling until the data has returned from memory, the scheduler can continue checking the other threads on the ready list. Some of those threads might hit the I-cache and can be moved to the active list, from where they can be executed.

The point of this two-phase approach is to ensure that the active list consists only of threads whose line in the instruction cache is *guaranteed* to be present. In order to guarantee this, each line in the instruction cache must also be extended with a reference counter. This counter counts the number of threads in the active and waiting list that require data from the cache-line. It is incremented in the I-cache check in the second phase and decremented when the pipeline performs a thread switch. Only lines with a reference count of zero can be evicted. If all lines in the I-cache have a non-zero reference count and another request comes in from the thread scheduler, it must stall. This is not a performance problem because the scheduler is an independent hardware process whose input queue is the *ready list*, which can always contain all threads. Also, if every line in the I-Cache has a non-zero reference count, then there must be at least as many threads on the active queue, waiting to be executed in the pipeline.

Figure 4.7 shows the organization of the thread scheduler. Whenever a thread switch occurs, the pipeline also reads the thread’s data from the thread and family table, such as its register frame pointer and its register counts.

In the thread scheduler, it’s desirable to move multiple threads from one list to another in a single cycle in two situations. The first situation is when a global register (which is visible to multiple threads in a family) has not yet been written, all threads in the family on that core can suspend on the register. This necessitates that the architecture supports a list of threads waiting on a register. When the register is written, all of these threads must be appended to the ready list. The second situation
Figure 4.7: The thread scheduler. Shown is the ready list (RL), the active list (AL), the activation process ($P_a$) and the various components that supply thread information and schedule events. The register file acts as the synchronization namespace.

exists because multiple threads can require the same line in the instruction cache. For this reason, the instruction cache must also support a list of threads waiting on each of its lines. When a cache-line returns from memory, all of the threads that were waiting on it must be appended to the active list.

Hardware FIFO buffers can not move an arbitrary number of items from one to another in a single cycle. Thus, to support these two crucial situations, the active, waiting and ready lists must be linked lists. The storage requirements for this are a single next field per support thread with a port for every process that can manipulate it. The lists themselves are implemented as a head and tail fields. A linked list of threads can be appended in its entirety to any other linked list in a single cycle.

The tail field is required for scheduling fairness constraints. In a simple linked list implementation, no tail field exists and items are appended to the front of the list (a FILO linked list). If this is applied to the active and ready lists, a few threads could continuously exist at the front of these lists, starving the threads behind it. As such, for at least these lists, threads should be appended to the back of the list, creating a FIFO linked list, requiring a tail field.

This form of buffer management using linked lists is, to the best of our knowledge, a novel approach to buffer management. Probably, however, because it is uniquely suited to the situation. This use of a linked list only works if there is a bounded size of items (in this case, threads), where every item can only be in one state at a time. Items in the same state can then be part of the same linked list.
Chapter 4. The Core Architecture

4.9.1 Control stream

The architecture was designed to use dynamic thread switching, rather than static thread switching, i.e. interleaving. A thread switch occurs in three situations: when a register is read that is not full, on a branch or jump, or when execution reaches the end of a cache line. The former requires a thread switch because the thread cannot continue until the data has become available; the thread identifier is saved in the register so the thread can be woken up when the register is written. The latter two situations cause the program counter to (possibly) proceed to a cache-line that is not in the I-Cache. Switching to another thread in these cases avoids a control hazard, saving the cost of a pipeline stall while the cache-line is being fetched, as well as the cost of branch prediction logic. Additionally, this technique has the beneficial side effect of fairness. Every thread will yield at worst once per cache-line, giving the other threads a chance to run.

However, two of the three situations are not known until later stages in the pipeline. Only the situation where the thread’s program counter leaves the cache-line can be identified at the very beginning of the pipeline. A branch is not identified until the decode stage and an empty register is not identified until the read stage. To avoid the overhead of flushing the pipeline in these two situations, the compiler can tag instructions that should cause a thread switch. This tag is encoded in a control instruction, which is stored in the so-called control stream, analogous to the normal instruction stream. Currently, only three kinds of 2-bit control instructions exist: continue, swch and kill. The former is the default when no tag is specified in the assembly source code and has no effect on thread control. swch causes a thread switch and kill terminates the thread after executing its tagged instruction.

To avoid maintaining two program counters and two I-cache accesses, the control stream is interleaved with the regular instruction stream. The 2-bit control instructions are packed into control words, of equal size to the architecture’s instruction word. The first instruction in every cache-line is then interpreted as a control word, where each control instruction in it maps to an instruction in the cache-line. This mapping is illustrated in figure 4.8. This mapping works for any RISC-like architecture with

![Figure 4.8: The control stream. At the bottom is the 512-bit cache-line, containing 16 instruction words. The first instruction word in each cache-line is the control word, containing the 2-bit control instruction (CI) for every instruction in the cache-line. The first CI is unused as it maps to the control word itself.](image-url)
fixed-size instruction words where the number of instructions in a cache-line is at most the number of control instructions that fit in a single instruction word. 

With this scheme, only a single program counter has to be maintained per thread, only a single I-cache line has to be fetched and the fetch stage can read both the instruction and the control instruction from the same I-cache line and, if the control instruction indicates so, perform a thread switch after the fetched instruction.

The compiler strategy for tagging instructions with `swch` should be a conservative one, considering that thread switches create no bubbles in the pipeline. In general, if an instruction can cause a bubble in the pipeline, it should be tagged with a switch. In the best case, it saves the bubbles that would have been caused by the flush. In the worst case, nothing is lost as the thread is simply rescheduled onto the ready queue and continued at a later time.

Note that although `kill` is implemented as a control instruction, it can also be implemented as a normal instruction. As a control instruction, it saves a cycle per thread, which can add up to a significant percentage for large families of small threads. However, should architectural constraints demand that control instructions be reduced to a single bit, `kill` can always be implemented as a normal instruction.

### 4.10 Instruction cache

The instruction cache is an N-way set associative lock-up free cache, allowing for efficient management of concurrent threads that require an instruction fetch without blocking the pipeline or cache.

To support the split-phase memory loads, the I-cache lines have four possible states instead of the normal two (valid and invalid): the empty state indicates that the entry is not being used. The loading state indicates that a read request has been sent to the memory; the line has no data. The invalid state indicates that the line has been invalidated by the memory hierarchy above this cache, but still has threads that are waiting on the line; the line cannot be cleared until the pending threads have run. Lastly, the full state indicates that the entry is present and contains all data for a cache-line. Figure illustrates the states and their changes for a cache-line entry.

The cache lines are also extended with additional fields that store the head and tail to a linked list of thread identifiers; the waiting list for that cache line. This list is initialized when a line is first fetched from memory. All threads that require the cache-line are appended onto this list by the thread scheduler (see section 4.9). When the cache line returns from memory, the list is appended to the thread scheduler’s active list.

To ensure that a loaded cache-line is not evicted before the threads that require it are executed, a reference counter is used to lock down the line. When the thread scheduler performs the I-Cache check for threads on the ready list, the reference

---

11 In order to support binary compatibility with the base ISA, we suggest that families can be marked as “legacy”, which disables the hardware logic for reading and skipping the control word.

12 Note that to avoid a situation where the line is forever kept present due to continuous instruction fetches, the thread scheduler must stall when it hits an invalid line.
counter is incremented. When pipeline no longer needs an I-Cache line by performing a thread switch, it decreases the reference counter. This may mean that some threads on the ready list cannot fetch their I-Cache line because no lines can be evicted until some threads are switched out of the pipeline. However, if the I-Cache is large enough, then the number of threads that can fetch their cache line and keep the pipeline busy will hide the latency of these fetches.

4.11 Data cache

The data cache is an N-way set associative lock-up free cache which uses the destination register as the miss information/status holding registers (MSHR) instead of defining a separate and dedicated register file for that purpose. Thus, when a memory load is pending, the destination register holds the information required to complete the load, such as the byte range in the cache-line that should be copied into the register, whether to sign-extend the data, and so on. A reference to the register is stored in the cache-line that it requires data from in the form of a per-line linked list of registers called the pending list.

This mechanism allows an outstanding memory read to every register without blocking the pipeline or cache at the expense of a head field in each D-cache line. Cache misses are handled by storing the register index in the cache-line’s head field and returning an “empty” value with the pending memory load information, and the old contents of the head field. This value is then stored by the pipeline as if it were the normal result. This technique constructs a linked list of registers without additional ports on the register file. When the data returns from memory, the index of the cache-line is put on a “service” list that the D-Cache will use to service the pending reads one-by-one; it reads the register pointed to by the line’s head field register, updates the head field with the value from the register and writes back the data from the cache-line as indicated by the pending load information from the register. If a thread has suspended on the register, this writeback will act as the synchronization.
event and wake it up (i.e., append the thread identifier stored in the registers to the scheduler’s ready list).

To support the split-phase memory loads, the D-cache lines has four possible states instead of the normal two (valid and invalid): the empty state indicates that the entry is not being used. The loading state indicates that a read request has been sent to the memory. Note that in this state, the entry may or may not have data in it; writes from the pipeline and writes snooped from the memory may be written to the cacheline. The line has a bitmask indicating which bytes have valid data; these bytes can be used to satisfy subsequent memory loads requests even though the entire line hasn’t been loaded yet. The invalid state indicates that the line has been invalidated by the memory system, but still has has registers that need to be processed. It cannot be cleared until the pending read requests have been processed. Although the line can still be used to service new local read requests, it should not be used to service local writes. If a write to the D-Cache hits an invalid (or loading) line, the write must stall, for the following reasons:

- A write into a loading (or invalid) entry might violate the sequential semantics of a single thread because pending reads might get the later write’s data (WAR hazard). For instance, a read to location A misses the cache, allocates an entry and sets the state to loading. A subsequent write (from the same thread) to A hits the cache and writes its data into the entry. The read completes and processes the pending reads. The register now gets the data from after the write, instead of before, as the thread would expect.

- The write cannot simply write-through while ignoring the entry because subsequent reads should get the new data and not the old, so the entry must be updated.

- The entry cannot be invalidated and a new line allocated, because read responses from the memory bus hit cache entries on address, and then there would be multiple valid lines of the same address. Read responses cannot identify cache
lines by index, since this would put an unnecessary and undesirable strain on the design of the higher memory architecture, where it would have to track cache-line indices of reads, which are most likely different for each cache on the memory bus.

Lastly, the full state indicates that the entry contains all data for a cache line and no read requests are pending. The data in the entry can be used for reads and writes. Figure 4.10 illustrates the states and their changes for a cache entry.

As a final remark, note that there is a possibility that the cache line is fetched from memory before the memory load instruction that caused the miss has advanced to the writeback stage. When the data cache tries to read the register identified by the line’s pending list for the pending load information, it will contain invalid data. To ensure a correct resolution of this scenario, the data cache, when reading this register, must stall its processing of the returned line if the register is not in the empty state, or without correct information in the MSHR fields.

### 4.12 Choice of ISA

When implementing an architecture with the microthreading extensions, a choice for a base architecture has to be made. Although the microthreading extensions can in principle be applied to any von Neumann-like instruction set, there are several common features from popular instruction sets which complicate the integration with microthreading:

- **Delay slots.** Found in the SPARC [84] instruction set among others, delay slots exist in a processor pipeline without hardware threading to reduce or remove the cost of pipeline flushes on branches (control hazards). Supporting these in a microthreaded pipeline would require special handling of thread switches on and around branches. It would also raise issues with backward compatibility and execution semantics when the delay slot lies across a cache-line boundary. Delay slots are especially useful in architectures without hardware threading, but given that the latter gives the processor another way to handle control hazards, having both in an architecture would be unnecessarily wasteful.

- **Predication.** Predicated execution, as found in the ARM [85] instruction set, exists to avoid pipeline flushes for branches that only branch over small pieces of code. Instead, the code is marked with a predicate that disables the instruction through the pipeline if a condition is not met. Although this could be of use in a microthreaded architecture, it is not critical to the understanding of the core issues of microthreading.

- **Implicit operands.** Certain instructions in popular instruction sets like x86 [86] and MIPS [87] contain implicit general-purpose register operands. These operands are implicit because the instruction set encodes a certain convention regarding the use of these registers. For instance, the x86 instruction set defines a specific general-purpose register as the stack register, such that all stack operations modify this register. Other instruction sets often denote specific registers
as destination registers for branches or arithmetic operations. Microthreading relies in part on being able to compress the register set of threads to the amount they require. This feature is incompatible with implicit operands unless logic is added to remap these registers based on information supplied by the compiler (akin to the register count word described in section 4.5.2).

- Non-general purpose registers. Ideally, an instruction set only requires general purpose registers. Popular instruction sets such as the x86, MIPS, Power and Sparc require additional registers such as flags, link, count, divider or overflow registers that exist outside the set of general purpose registers. These registers are often additionally written to, next to the instruction’s main result operand. This requires either additional ports on the register file if these registers are mapped there, or extra fields of considerable size in the thread table. Supporting these would both complicate the implementation and distort the size required for on-chip data structures to be greater than strictly necessary to support microthreading.

- Complex operands. Certain instruction sets like the Sparc feature variable-sized operands. Depending on the instruction, an operand can specify 1, 2 or 4 consecutive registers. Coupled with the concept of I-structures, situations can arise where different parts of the operand are in a different state. To support this, the pipeline would have to be extended with significant logic to allow for multi-cycle reads and write from and to the register file, taking its mixed state into account.

Note that these issues do not prevent using microthreading, but they would complicate an implementation.

For the research in this thesis, and the implementation for the evaluation in chapter 7, a base instruction was needed that allowed the research effort to be focused on the issues surrounding microthreading itself, and not the interaction with details of the instruction set. Additionally, the instruction set should be supported by popular tools in order to simplify development and increase the appeal of the implementation to others in the community.

After considering these issues, we chose the DEC Alpha [88] instruction set as it featured none of these undesirable features and was supported in the open-source GNU Compiler Collection [89], allowing for the quick and easy creation of an assembler and linker specific to the microgrid.

### 4.13 Size analysis

The areas of the core data structures (family table, thread table, I-cache, D-cache and register files) as well as the memory data structures have been calculated using the C++ version of CACTI 6.5 [90], with settings as listed in table 4.1. The areas of the functional components (ALU, multiplier and FPU) have been calculated based on [91]. For the family table and thread table, the area for each field in each table was calculated independently based on the width of the field and the required ports.
Temperature: 350 K (76.85 °C)
Access mode: Sequential (tag first; then data)
Design: 0; 0; 0; 0; 100
Deviate: 60; 100000; 100000; 100000; 1000000
Optimize: None (use design and deviate)
Wire type: Global, 10% penalty
Data array cell type: ITRS-HP
Data array peripheral type: ITRS-HP
Tag array cell cell: ITRS-HP
Tag array peripheral cell: ITRS-HP
Interconnection projection: Conservative
Wire inside mat: Global
Wire outside mat: Global
Cache type: UCA (Uniform Cache Architecture)
Cache level: L2
Repeaters in H-tree segments: Yes
Add ECC bits: No

Table 4.1: CACTI settings for area calculations. Values given for design and deviate are: delay; dynamic power; leakage power; cycle time; area.

<table>
<thead>
<tr>
<th>Component</th>
<th>Area (mm²)</th>
<th>Access time (ns)</th>
</tr>
</thead>
<tbody>
<tr>
<td>D-Cache</td>
<td>0.160</td>
<td>28%</td>
</tr>
<tr>
<td>Int. Register File</td>
<td>0.080</td>
<td>14%</td>
</tr>
<tr>
<td>Flt. Register File</td>
<td>0.080</td>
<td>14%</td>
</tr>
<tr>
<td>I-Cache</td>
<td>0.069</td>
<td>12%</td>
</tr>
<tr>
<td>Family Table</td>
<td>0.050</td>
<td>9%</td>
</tr>
<tr>
<td>ALU</td>
<td>0.047</td>
<td>8%</td>
</tr>
<tr>
<td>Thread Table</td>
<td>0.046</td>
<td>8%</td>
</tr>
<tr>
<td>Multiplier</td>
<td>0.036</td>
<td>7%</td>
</tr>
<tr>
<td><strong>Total</strong></td>
<td><strong>0.568</strong></td>
<td><strong>100%</strong></td>
</tr>
</tbody>
</table>

Table 4.2: Area breakdown of a single core.

Using this method, the area of a single core has been calculated as 0.57 mm² with a 35 nm process. The breakdown of this area into its components is shown in table 4.2. The core has 32 families, 256 threads, 1024 integer and floating-point registers, a 2 kB 4-way set associative I-Cache and a 4 kB 4-way set associative D-Cache. The table also shows the calculated access times for each structure. With a combined maximum access time of 0.42 ns, each core can be safely run at 1 GHz.

### 4.14 Challenges and Contributions

The contribution made by this thesis in the most general sense is to have taken ideas from prior research and to identify and solve the many challenges in creating an independent, fully functional and efficient implementation. The prior research started with the design of a Dynamic RISC (D-RISC) processor [69] which also coined the term *micro-threads*. To effect dynamic scheduling in a D-RISC core, it introduced
split-phase memory loads and synchronization on registers where the continuation PC is stored in the empty register. Additional research fleshed this design out into a possible core architecture \[92\]. Up until then, threads were still created one at a time, but the concept of \textit{families} of threads was soon introduced into the architecture \[93\]. This also introduced the concept of globals, locals, shareds and dependents as logical views into a single register file. Soon afterward, early simulation results became available based on the Alpha 21464 instruction set, the design was fleshed out even more \[94, 95\] and concrete estimates were made on the cost of the required hardware structures \[96\].

Since then, the research described in this thesis has combined the previous work into a single, functional and simulable design with proven support for an actual software compiler stack enabling execution of parallelizable C-like programs on the architecture. This architectural design went through several iterative phases. Certain approaches were found through experimental validation to either not work or not be as efficient as desired. Due to the highly parallel nature of the architecture, deadlocks in hardware processes were found to occur which required a rethink of the design. Additionally, related research within the group such as compiler or operating system research gave insights into how the hardware should behave, which required a change in the latter’s design. All of the these issues indicate points where the design of the hardware had to change; sometimes minimally, but sometimes a more invasive redesign was in order.

The architecture presented in this chapter (and the next) represent the state of the design at the end of the process. As such, it gives the impression that things “just work” and fit together. However, to put this research in the proper context, below we will give an overview of some of challenges and design choices that appeared on the road to this architecture.

### 4.14.1 The control stream

Because of the dataflow synchronization via registers, it was very likely that an instruction that reads a register would find that register in the empty state. This situation, however, could not be identified until that instruction reached the Register Read stage in the pipeline, at which point the pipeline has to be flushed because the thread cannot continue. This wastes cycles and would not be a rare occurrence. We quickly figured that some sort of simple instruction was necessary to indicate to the pipeline fetch stage that it should move to the next thread. Obviously the instruction that reads the register could not be avoided, but switching to another thread after that instruction would avoid any more costly bubbles in the pipeline. This “switch” instruction had to be simple enough that it could be decoded at the Fetch stage, and not need to pass the Decode stage first.

The initial solution featured a special instruction that could be placed after every instruction which would make the pipeline switch threads. We discarded this solution because it negatively impacted performance by significantly increasing the code size in memory. This led us to realize that really only a single bit is required to indicate if the pipeline should switch or not. We briefly considered extending the instruction words with this bit, but that was deemed impossible due to alignment issues in memory and
the required compatibility with the base RISC instruction set. Before settling on the final solution, we considered placing these switch bits in a separate instruction stream, with its own program counter. This solution was discarded because it would overly complicate the instruction fetching logic and the I-Cache, which would potentially require fetching from two different cache-lines every cycle. I-Cache miss handling would become more complicated.

The contribution made in this thesis was, as described in section 4.9.1, a solution where the “switch” bits are packed into the first word in a cache-line. This was deemed the best solution as it kept all data related to an instruction in the same cache-line, kept memory overhead to a minimum and maintained compatibility with the base instruction set: legacy instruction streams could be flagged in which case the fetch logic would not treat the first instruction word as a control word. This solution does require future architectures to have a cache-line size an integer multiple of the minimal cache-line size, but in our opinion this was not an objection.

Additionally, in the case of the Alpha instruction set with its 32-bit instruction words, we only used half the bits in the 32-bit instruction word to annotate the 16 instruction words that fit in a 64-byte cache-line. We chose to use the extra bit per instruction to encode the “kill” bit, an optimization which allowed extremely small threads (e.g. families that implement loops with an extremely small body) to avoid the extra cycle per thread just to end the thread.

4.14.2 Thread scheduling

The previously proposed versions of the architecture had a so-called “continuation queue”; this was a simple FIFO buffer which contained indices for the threads on that core that should be executed in the pipeline. The pipeline, when a thread switch was detected (see above), would fetch the first thread off this queue, fetch its I-Cache line and execute it until a thread switch was encountered or the thread was terminated. This mechanism had a flaw which was easily spotted when analyzing the results of experiments: this scheduling did not take the I-Cache into account; in the case of an I-Cache miss, the pipeline would stall and wait before executing the thread, despite the fact that other threads in the continuation queue might not miss the I-Cache.

The contribution of the work described in this thesis was to fix this flaw by splitting this continuation queue into a ready queue, an active queue and a hardware process that checks the I-Cache (see sections 4.9 and 4.10) for threads in the ready list and in collaboration with the I-Cache, puts threads on the active queue when their I-Cache line is present, guaranteeing that the pipeline will always hit the I-Cache when fetching threads off the active queue.

This situation is one of the main cases where the benefits of the linked-list thread management comes into play. By using a linked list, threads that miss the I-cache would be efficiently linked into a per-cache-line linked list. Once the cache-line was retrieved from memory, this entire list could be efficiently appended to the active list.
4.14.3 Thread distribution

Initially, the architecture featured a global scheduler that contained every created family. It used a broadcast bus to distribute the threads across the different cores. As noted in chapter 1, global communication across a chip is problematic with increasing clock frequencies, so having a global broadcast bus and single arbitrating component was quickly identified as a bottleneck to the scalability requirements, but a proper solution for its replacement required many iterations in the design cycle. Hence a major contribution of the work undertaken in this thesis was the design, implementation and evaluation of a scalable means for the distribution of threads to different cores.

The first solution that we tried was to distribute the global scheduler by giving every core a copy of it and using a circular daisy-chained network with a token-based protocol to implement global exclusion to create and distribute families. This token protocol was non-trivial and went through several iterations of priority rules and fast-forwarding circuits. Eventually, we realized that a global token was still a non-scalable solution to a system which would be creating families in parallel.

Our next solution removed the token entirely, giving every core free reign to create and distribute families to its neighbours. This quickly resulted in deadlock situations where neighbouring cores both needed to allocate families on each other, but both of them just allocated their last entry. Without a global token to arbitrate these issues, we tried to mitigate this problem. We statically split the family table into global and local families, the latter would only run on the core itself and not require allocations on other cores. Several design iterations went through optimizing this strategy.

Additionally, to avoid the delays and unnecessary allocations of traversing the entire chip when allocating a global family of only a few threads, we statically divided the chip into distinct “rings” of cores with a delegation network between the roots of the rings. For instance, a chip of 256 cores could have one 128-core ring, one 64-core ring, one 32-core ring, and so on. A thread wanting to create a family could then choose what size ring to put the family on. The family’s child families would stay on that ring of cores until an explicit delegation to another sized ring was made. The problem with this approach was that the chip lost its generality; it had to be tailored towards the applications that were expected to run on it; if there were applications that required a lot of large or small rings, a lot of the computational power on a chip with the “standard” configuration would be wasted.

Another contribution was to analyse and optimise this situation, by removing a feature rather than solving it; in one of the biggest and most impacting redesigns of the architecture we replaced the circular daisy-chained network with a single a daisy chained network and a full core-to-core delegation network. The static rings were removed and would now be virtually partitioned out of the single long chain of cores on the chip (see sections 5.1 and 5.2). To avoid the allocation deadlock scenario in the case of exhausting family table entries, we implemented the fallback mechanism mentioned in section 4.5.4.
4.14.4 Dependent families

Dependent families—that is, families with one or more so-called *shareds*—have been a recurrent feature in earlier work which were implemented by mapping the output registers of one thread onto the input registers of another thread. This presented obvious problems when these threads were not on the same core when a family is distributed across multiple cores. For this case, we initially developed a protocol between neighbouring cores that exchanged these data. The pipeline would recognize when a register was read or written that needed to communicate with the neighbouring core and trigger special action in the network component.

Deemed necessary, this protocol suffered from a lot of pitfalls and gotchas. For instance, whereas the creation of threads on a single core was ordered (i.e., thread \( i + 1 \) is created after thread \( i \)), this was not true between cores. The protocol became one where every case and combination of the creation order had to be accounted for. The output thread could write before the input thread had read, or even before the input thread was created. The input thread could read before the output thread had been created. The output thread could write before even the family entry on the next core had been allocated. And so on. These problems required many design iterations of different synchronizing messages, discarding messages and sanity checking. We had to iterate on this design every time another, previously unconsidered, issue popped up.

In the end, we choose to completely sidestep these problems by restricting dependent families to a single core. The rationale behind this decision, as discussed in section [4.5](#), was that dependent families run largely sequentially anyway, and there were algorithmic solutions to the cases where it would otherwise make sense to distribute them. This single decision reduced a lot of the complexity of the cores and network protocols.

4.14.5 Family creation

When creating a family of threads, it had to of course be possible to parameterize the family creation itself (i.e. how many threads, the program counter, and so on). The previous approach was to pass the memory address of a structure called the Thread Control Block (TCB) to the create instruction. This structure contained all the information about the family needed to create it. However, the related compiler research soon flagged this as a problem because the TCB could not be statically allocated in memory when the parameters depend on runtime variables, and had to always be filled out, even if most of the properties were the same across many families. From the hardware point of view, in-memory TCBs presented another layer of complexity in creating a family: first the TCB had to be fetched from memory.

Another problem with this approach flagged by the compiler research is that it was incompatible with C-style linkage. In the case where a thread function is defined in a library, the caller (or family creator) cannot know from the definition of the thread function how many registers it requires; a fact which had to be known to the hardware when creating a family.

Therefore, another contribution of the work described in this thesis was to solve both of these problems. We completely redesigned the family creation process and
split up family creation into three steps: allocate, setup and create; each implemented via one or more special instructions. This made compilation for the microgrid significantly easier and avoided dynamic memory allocation during dynamic family creation. By having separate instructions for setting each of the parameters of the family, it was possible to omit them in the case where the default value was sufficient. We choose the default value of the parameters such that it would create a family of a single thread; this would allow for low-overhead creation of functional concurrency. The other case, families with lots of threads, could afford the extra cycles of overhead to set up the full family parameters.

Additionally, instead of the caller providing the register information, the callee had to provide it: we choose to store the thread’s desired register context information at the start of the body of the thread code in the form of a register count word (see section 4.5.2), which would be read by the hardware’s family creation process.

### 4.14.6 Family parameters

As described in chapter 3, a family has ties to the creating thread in that the latter supplies the values of the globals, supplies the initial shareds and receives their final output. Initially this was implemented in the spirit of the shared/dependent register sharing within a single family: a family’s globals mapped on a section of the creating thread’s locals, and similarly for the first and last of the family’s shareds. The creating thread simply specifies a base register within its local registers and after family creation, can simply write to the right local registers to set the arguments. In the case of a delegated family which ran elsewhere on the chip, a proxy entry on the creating core was allocated so these registers could be forwarded across the network.

This technique was ultimately replaced because of several problems: firstly, the compiler research indicated that this technique was insurmountably incompatible with existing compiler assumptions; it would have taken a large amount of work to get the compiler to understand that between two points in the code, certain registers had a special meaning. Next to this, it would also reduce the operational size of the register window of the creating thread while the family was active. Secondly, this proxy entry added additional complexity to the core. Both issues limited how many families a thread could create at any given time.

Another contribution was therefore to replace this register mapping technique (where the global and shared data are pulled from the creating thread) with a technique where the creating thread pushes the values to the created family via special instructions. This approach completely decoupled the created family from the creating thread and had several originally unintended benefits: it allowed threads created by one thread to be synched upon in another thread, and it allowed for detached families, where the family is never synched upon. These consequences opened up additional avenues of operating systems and compiler research.
Chapter 4. The Core Architecture

4.15 Summary

In this chapter, we described the architecture of a single multithreaded core that can be composed into a System-on-Chip. It described the various components necessary for the bookkeeping information of the thread contexts: the thread and family table (section 4.3), containing the contexts of a single thread, and a (finite) set of threads, respectively. It is exactly this concept of families that allows programs to express more concurrency in software that than there is available parallelism in hardware. In sections 4.5 and 4.6 we described how the synchronization on these threads and families works and in section 4.8 we described a possible solution to managing thread-local storage with such a large number of thread contexts managed in hardware.

We also described the thread scheduler (section 4.9), which uses on-chip linked lists of thread contexts to ensure low-overhead bulk migration of thread states from one state to another; a technique which, to the best of our research, has not been applied in architectures before. We described the design of the I-Cache (section 4.10) to support these linked lists.

Additionally, we described the register file (section 4.4), which is larger than normal to accommodate for the relatively large number of thread contexts, and with extra state bits that, together with the modified pipeline, allows for dataflow-like synchronization on individual registers. Coupled with a control stream that is efficiently multiplexed with the instruction stream, this allows for extremely efficient pipeline utilization even in the case of long-latency operations such as uncached memory loads, I/O or FPU operations. And we described the design of the D-Cache (section 4.11), which requires modifications to be able to support these uncached memory loads.

The design in this chapter is general enough for any kind of RISC-like architecture. However, we had to choose a specific instruction set to base our emulated implementation on. In section 4.12 we justified the choice of this instruction set.

Finally, in section 4.13 we analyzed the size of an expected typical configuration of this core architecture using the industry-standard tool CACTI.

Although energy efficiency has not been explicitly mentioned nor quantified in this chapter, it was the core motivator behind the design. The core was designed to minimize non-scalable hardware structures and avoid wasting energy by performing unnecessary work. It is the reason why is there no cache prefetching, why threads are queued in a linked list per I-Cache line on an I-Cache miss, why the pipeline uses compiler-assisted thread switches in favor of branch prediction or predicated execution, and why long-latency operations have been implemented as split-phase operations with dataflow semantics on registers.

This architecture represents the outcome of the main contribution of this thesis: the identification, analysis and solving of inefficiencies and bottlenecks, collaboration with peer researchers and the combining of multiple ideas and proposed techniques into a single, working, scalable, efficient and deadlock-free system-on-chip design. In particular, a key contribution that is part of this is a novel method to efficiently manage collections of hardware threads in various states via linked lists implemented entirely in on-chip static memory.

The architecture presented in this chapter is directly reflected in publications P.1, P.4, P.6, P.7, P.8, P.9, P.10, P.11, P.12 and P.13. As in all architecture development,
the resulting system is a collaboration between system architecture, compilers and operating system development. As a result, we have been directly involved in the refinement and reuse of the architecture presented in this thesis in all these publications, where we made significant contributions on the architecture design and interface.

The next chapter will show how this single core can be composed scalably into a system-on-chip.
Chapter 5

The Chip Architecture

“The whole is greater than the sum of its parts.”
- Aristotle

The previous chapter described the main architectural components of a single core in the microgrid. This chapter describes the chip-wide architectural improvements that are a result of the research in this thesis. This is primarily the design and implementation of places. The on-chip network, distributed memory and I/O systems, despite not being the primary target of the research, are also described since they are all required for a working system.

With the exception of the I/O system (see section 5.5), all content in this chapter has been our novel contribution, designed specifically for this architecture. Like in chapter 4, the designs described in this chapter were based on existing concepts, but have been adapted to new research directions and fleshed out to fit together with the core architecture. The memory system in particular (see section 5.4) was inspired by earlier work on distributed cache memory systems [97, 98, 99], but had to be improved and, in places, redesigned to eliminate deadlocks that arose in practical cases. We also re-implemented the memory system to be compatible with the simulator architecture (see chapter 6) as the original research had an experimental implementation in an incompatible framework.

5.1 Microgrid

The microgrid is a system-on-chip composed of multiple cores of the kind as described in chapter 4. Figure 5.1 shows an example microgrid configuration consisting of 64 cores, 32 FPUs and a distributed shared-memory network consisting of 16 caches configured into 4 rings, each hooked up in a ring to a single off-chip DDR channel. Two cores are extended with an I/O controller allowing off-chip communication with devices. The 64 cores are connected via a single daisy-chained network that is laid out in a Moore curve. This type of curve was chosen because it is a space-filling curve that preserves locality around the L2 caches and can easily be scaled to grids of more or fewer cores. The mesh network is not shown but lays on top of this diagram,
Figure 5.1: Overview of a microgrid consisting of 64 cores (blank squares), 32 FPUs (F), 16 L2 caches, 4 directories (D), a single root directory (R) with an off-chip DDR channel, and two off-chip I/O channels. The single gray Moore curve connecting all cores in a daisy-chain is the link network. Not shown is the mesh network connecting all cores.
5.2 Places

As the number of cores on a processor increase, applications will move from time-sharing a chip to space-sharing [100]. This means that instead of running multiple applications (e.g., threads or processes) on the same core(s), applications will be mapped to entirely separate cores. The cores in a system will become a resource like memory is today, where application can allocate, use and free chunks of cores. In the microgrid, this is supported by automatically distributing families of any size across any number of cores.

Programs can have different requirements on the number of cores for a piece of computation. Some algorithms can parallelize well and can take many cores, whereas some might see reduced benefit for larger number of cores. In order to cope with both situations, a microgrid must be able to serve requests for different number of cores, from 1 up to hundreds of cores. The result of such an allocation of cores is called a place. It is a conceptual entity, implemented as a set of cores, that can be the target of a create. The cores in a place can be considered as a linear chain of cores, where each neighbouring pair is connected via a special network, in order to allow neighbour-to-neighbour messages relating to family management to be transmitted quickly, avoiding the routing mesh network. As such, all the cores in the microgrid are daisy chained via a linear bi-directional “link” network. Any consecutive sequence of cores on this link network (subject to some constraints, see below) can be used as the target for the creation of a family of threads. The cores use the link network to evenly distribute the threads in the family across the specified cores as described in section 4.5.

The software is responsible for constructing the so-called place identifier for the range of cores. To allow compilers and programmers to efficiently distribute pieces of work across a range of allocated cores, it is desirable to support a method for manipulating this identifier without expensive library calls. However, in a naive implementation of having the range offset and size side-by-side in a single value, this breaks the constraint that application code must be compatible with older and newer versions of the architecture.

This problem was solved by using a special and novel encoding of the range in the place identifier. Figure 5.2 shows the contents of the place identifier, which is meant to fit in a single register. It contains a combined Address/Size field and a capability field. The latter is a security token that is matched with a special register on the destination cores to ensure that the software can only create to the cores it has been
Chapter 5. The Chip Architecture

<table>
<thead>
<tr>
<th>Message</th>
<th>Description</th>
<th>Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>allocate</td>
<td>This message attempts to allocate a family context across a range of cores. If the resources are not available, this message can suspend in a queue until they are.</td>
<td>B</td>
</tr>
<tr>
<td>configure</td>
<td>This message configures an allocated family context.</td>
<td>NB</td>
</tr>
<tr>
<td>create</td>
<td>This message starts thread creation for a configured family context.</td>
<td>NB</td>
</tr>
<tr>
<td>put</td>
<td>This message writes a value into a register context of a created family context.</td>
<td>NB</td>
</tr>
<tr>
<td>sync</td>
<td>This message sets up a sync-notification writeback in a created family context.</td>
<td>R</td>
</tr>
<tr>
<td>get</td>
<td>This message reads a value from a register context of a synched family context.</td>
<td>R</td>
</tr>
<tr>
<td>detach</td>
<td>This message marks a created family created as detached.</td>
<td>NB</td>
</tr>
</tbody>
</table>

Table 5.1: List of network messages on the microgrid with their types: Blocking (B), Non-blocking (NB) or Returnable (R).

The address and size of a place on a grid of $N$ cores can each be represented by a value of $A = \lceil \log_2 N \rceil$ bits. However, in the place identifier, these two values are packed together in a field of $A + 1$ bits. The encoding is defined as follows:

$$\text{Encode}(\text{Address}, \text{Size}) = \text{Address} \times 2 + \text{Size}$$

This encoding requires that $\text{Size}$ is a power of two and that $\text{Address}$ is a multiple of $\text{Size}$. The former requirement ensures that the hardware can calculate thread distributions with shifts instead of complex divisions. Combined with the latter requirement, it also enables this encoding, which allows for manipulation of a place identifier by software without implementation-dependent knowledge or library calls. For instance, given a place identifier $X$, it is possible to create a new place identifier $Y$ that specifies the upper half of $X$’s range of cores as follows: $Y = X + (1 << \text{bitscan}(X))/2$. The bitscan is an operation that returns the index of the least significant ‘1’, which can be implemented as a simple single-cycle operation using a barrel shifter in the integer ALU, which is already available in most ISAs including the one used for the evaluation in this thesis (the Alpha ISA). Given the encoding scheme, $1 << \text{bitscan}(X)$ will extract the size from the place identifier. By adding half the size to the place identifier, a new place identifier is formed that defines a range that is half the size of the original, and starts at the original’s address, offset by half the size. Note that none of these operations require knowledge of $N$, the number of cores in the grid, or $A$, the number of bits that the $\text{Address/Size}$ field is encoded in.
5.3 Network

The microgrid as described above is connected via a network-on-chip. There are several messages sent over this network, listed in table 5.1. Blocking messages are messages that can be queued on their destination, awaiting some condition until they can be handled. Non-blocking messages are guaranteed to be handled and destroyed within a finite time of being received. Returnable messages are non-blocking messages that can create a non-blockable message that is returned to the sender.

Any network implementation that is to be used on the microgrid must meet the following criteria:

- Order-preserving. Some of the messages sent to the same family should not overtake another on the network. For instance, if the `detach` message overtakes the prior-issued `sync` message, the family could be detached and cleaned up before the sync message is handled, resulting in unintended behavior. As such, the network must guarantee that messages sent between cores $X$ and $Y$, are handled at $Y$ in the same order as they are sent from $X$.

Note that strictly speaking, this constraint allows for network with adaptive routing, allowing messages to arrive at cores in a different order. A per-core re-order buffer can be used to ensure the constraint, as long as the buffer is big enough to hold all out-of-order messages.

- Deadlock freedom. Although this constraint is trivial, its implementation is less so. Because the `allocate` message can suspend on the target core, this could lead to a deadlock situation in a simple network. For instance, if a core is full, and one or more cores send several `allocate` messages to that core, the allocates will queue up (eventually across the network) waiting for the currently running families to terminate. However, these families may depend on values yet-to-be-written by other cores with `put` messages. Since these messages cannot reach the core, deadlock occurs.

These criteria have been met in the architecture in the form of a mesh network with XY-routing and two virtual channels, one for the `allocate` (i.e., blocking) messages and one for all other (i.e., non-blocking) messages. With these virtual channels, the `put` message in the above example can reach its destination core over the non-blocking virtual channel whereas the `allocate` message can fill up the other virtual channel.

However, the network solution is only a partial solution to deadlock freedom. For instance, the amount of `allocate` messages could be so high, that the virtual channel’s transmit buffer on the source core fills up. In this case, the pipeline cannot send another `allocate` message, resulting in a pipeline stall, which might cause the `put` to still never to be sent. Therefore, a solution in the pipeline must be adopted as well, in case the `allocate`’s send buffer is stalled at that point. Since it can be safely assumed that the `put` message will originate from a thread that is not sending one of the offending `allocate` messages, a solution can be adopted where a full send buffer for `allocate` results in a thread switch instead of a pipeline stall; the thread is enqueued on the back of the active queue and will be run again later. This way, the thread that has to issue the `put` message will get a chance to run. Although this
Chapter 5. The Chip Architecture

solution is a weak form of busy-waiting, and goes against the architecture’s principle of energy efficiency, it solves the rare situation outlined above and is cheaper to implement than another thread suspend and wakeup signal.

5.4 Memory

In a system-on-chip of hundreds of cores, running heterogeneous applications, a simple memory bus to off-chip memory becomes infeasible. Current and future manufacturing technologies cause signals to reach less and less area of a chip within a single cycle. To create a memory bus in such situations would require implementations of many cycles for a single transaction. Coupled with the single-writer arbitration model, such a memory bus would quickly become a major bottleneck in the architecture.

In principle the microgrid cores could be connected to any memory network that supports split-phase transactions, request pipelining and store acknowledgments. However, opportunities exist to optimize the memory protocol when it is known in advance that all cores are microthreaded cores. The rest of this section describes such a custom memory design developed specifically to evaluate the core and chip design presented earlier in this thesis.

Ideally, a memory network with local interactions must be used, preferably with distributed memory or caches to keep data close to the cores that uses it. The implemented memory is based on a hybrid between a cache-only memory architecture and shared memory architecture, which has been further refined and optimized to match the memory consistency demands of the architecture. The execution model of the architecture allows for a relaxed consistency model from the memory implementation.

The memory network implements a shared memory view rather than a distributed memory view for simplicity of use. Likewise, it implements a form of relaxed global consistency. While weaker than sequential consistency, it happens to be stronger than the consistency model exposed to programmers. We chose to keep the exposed model weak so as to allow for future work on additional memory optimization.

The execution model defines a form of thread-sequential consistency. Writes made by a thread are guaranteed to be consistent for all memory operations following the write in the same thread, however, they are not guaranteed to ever become visible to other threads, even in the same family. Writes made by a thread only become visible and consistent to the parent thread after the latter has synchronized on the family that the thread belongs to. In other words, from the point of view of the parent thread, it is as if the writes made by the threads in the family it created all happen at the point where it synchronizes on the family.

The memory model that has been implemented uses a stronger consistency model, a form of global consistency, where it is guaranteed that writes made by a thread will, within a finite time, become visible to all threads in the system. This consistency model has been chosen because it was easy to implement and satisfied the consistency model of the execution model. That is, programs that behave correctly under the execution model’s memory consistency model, will also behave correctly under the implemented memory’s consistency model.
Figure 5.3: Example connectivity diagram of the memory network showing 12 caches (circles) and 3 directories (double squares). The root directory also serves as memory controller for off-chip DDR memory. Not shown are the buses connecting each cache to its memory clients.

5.4.1 Overview

The memory architecture consists of a set of caches which hold the data in the network. Each cache serves one or more memory clients (e.g., cores) via a bus. The number of clients on a single bus depends on the used technology and frequency. The caches are connected in a hierarchy of rings, with directories providing the path between higher and lower rings. An example configuration is shown in figure 5.3. The (root) directories hold an index of all the caches in the ring below it. Note that to be able to do this, they must have as many sets as each of the caches in the ring (assuming homogenous caches) and an associativity that is the sum of the associativities of the caches in the ring.

5.4.2 Protocol

Cache-lines are introduced into the system by the root directory when a read request misses the index (i.e., the requested cache-line is not in the memory system). When a read request passes by a cache that has the cache line, a copy is made and sent back to the requesting core. When a read response arrives at the cache that sent its request (where the cache-line entry is locked in the mean time), the cache-line is stored and broadcast on the bus to the memory clients.

When a store is issued from a memory client and the address misses the cache, the request is handled as though it was two separate requests; a load followed by a store: a cache-line entry is allocated and the entire cache-line is requested. Then, the write is handled as write to a loading cache-line: the write is stored in the cache-line
and the dirty bitmask is updated to reflect the changes to the cache-line. When the cache-line returns with the data, the data is masked by the dirty bitmask such that only the unwritten parts of the cache-line entry are filled with the newly arrived data.

To effect global consistency, a store generates an update message on the memory network, which travels around the ring and updates all copies of the cache-line that it encounters. However, since such a message costs energy and bandwidth, it would be worth avoiding this if possible. The message to update all other copies can be omitted if the cache-line that it writes to is the only copy in the system. Naively, this can be established via an exclusive flag on the cache-line that is set when the cache-line is introduced into the system, and cleared when a copy is made of it. However, this ignores the fact that through evictions, several copies can merge back into one copy. With an exclusive flag, it would be impossible to know when all made copies have merged back into one. Instead, a token counting protocol is used. Each cache-line starts out with $N$ tokens when it is introduced into the system, where $N$ is equal to the number of caches in the system. A copy can only be made of a cache-line if it has two or more tokens; at least one for the cache to keep and at least one for the copy. When merging an evicted cache-line, the tokens are added. The result of this protocol is that a line is the only copy if and only if it has all the tokens, in which case writes do not have to be propagated onto the memory network.

Concurrent stores to the same address by different caches are not a problem; both stores will generate an update message on the network, which update each others cache-lines. The end result is a state where both cache-lines contain different data, but this is allowed by the memory consistency model of the execution model, which states that concurrent stores to the same location result in non-deterministic results. In this case, different parts of the chip will get different data when reading the location after the stores. When one of the copies gets evicted, the merging cache can choose to discard either copy of the location without violating the memory semantics.

Note that traditionally, writes to a shared cache-line invalidate all other copies such that the originating cache-line is the only exclusive copy left. This mechanism was also considered in the implementation, but raised issues. As is often the case, the problem is concurrent writes. In this case, two cache-lines would both try to invalidate the rest, including each other. One cache-line must be given priority and there are several ways to accomplish this. Caches could be numbered, and lower-numbered caches have priority over higher-numbered caches. Alternatively, a priority token could be introduced, where the holder of the token has priority. Either way, the losing caches need to re-fetch the cache-line in order to write to it. If this happens for multiple caches, it could significantly degrade performance of the memory network. This mechanism is only required if memory writes require strict ordering, but since this is not the case in the microgrid, a more relaxed solution could be implemented. This is why it was chosen to have a fairly quick resolution of writes locally and merge on evictions.

5.4.3 Deadlock freedom

Brookes and Roscoe showed in [101] that in a ring network, deadlock can be avoided by implementing a local rule that ensures that messages are only inserted into the
ring network when the buffer they are inserted into has at least two entries available. This guarantees at least one entry remains available for accepting messages from the previous node, thus keeping the messages in the network moving.

However, in a hierarchical network there is no longer a single ring network, and as such, guaranteeing deadlock freedom requires additional rules. If the local rule is only applied to lower cache ring networks, a situation can arise where all buffers in the top-level ring are filled up with messages and unable to proceed, even though the local rule at the caches was never violated. However, simply applying the local rule on the top-level ring as well does not solve the deadlock problem either as a situation can occur where both the top-level and a lower level ring are full except for one buffer entry and neither can put a message on the other ring.

To guarantee deadlock freedom, the hierarchical ring network must be reduced to a single ring network. This can be done if the directories are treated as shortcuts across a single ring. Figure 5.4 shows the example memory network from figure 5.3 and its “flattened” version. By treating the directories as shortcuts across the single ring, deadlock freedom can be ensured by applying the message insertion rule to the flattened ring and treating the shortcut path as an optimization only. Thus, if a message arrives at a directory and the directory indicates the cache-line does not exist in the ring below it, the message can try to shortcut and stay on the top-level ring. However, if the buffer in which the message would be placed does not have at least two entries free, the directory must fall back to the non-optimal choice of sending the message around the lower ring (i.e., continue on the flattened ring).

This flattened ring network is the worst-case path that any message in the network can take. For instance, figure 5.4 also shows a shortcut at the root directory. This shortcut exists for messages that, according to the root directory, have already
been loaded from memory. All other messages must traverse the queues to and from the external memory in order to read their data. Therefore, those buffers must be considered part of the flattened ring network and the shortcut an optimization. As a result of this, it is possible that a message that should have been forwarded on the top-level ring, must traverse the memory queues (although no actual operation to the memory has to be performed).

By applying this technique, deadlock freedom can be guaranteed on the hierarchical ring network.

5.5 External I/O

The microgrid has up to now been described as a self-contained system consisting solely of a multi-core processor and a memory network. Naturally, communication with external devices is required for interaction with the user, transfer of data to and from the system, or both. Contemporary approaches to I/O focus on integrating the I/O address space with the memory space, reserving a certain part of the memory’s global address space, and directing memory loads and stores onto the I/O bus usually at the point of the memory controller that is connected to external RAM.

In a highly concurrent system, such a design wastes valuable external memory bandwidth on I/O bandwidth, whereas the two are, conceptually, fully independent. In the microgrid, a solution has been adopted where designated cores in the microgrid have connections with an I/O bus. Thus, the mapping from address space to I/O space happens at the core level, rather than the chip boundary. The benefits of this approach are twofold. First, it is possible to trivially parallelize I/O operations by connecting multiple I/O buses to multiple cores, increasing I/O bandwidth to multiple devices. Second, any data read from I/O devices is injected into the memory system from the bottom, at the L2 cache level, rather than from the top. Due to the nature of the memory system, this data will remain local to where it is has been read, allowing system services to cluster applications that use the data around the cores that read the data. Ideally, the data would never have to leave the local cache ring.

These I/O cores have special logic in their pipelines to distinguish memory operations from I/O operations, typically by address mapping. The I/O operations are forwarded to an I/O controller that is attached to, or integrated with, the core. This I/O controller is responsible for taking the split-phase I/O operations and serializing them onto the I/O bus. This approach is the solution to the problem where contemporary I/O protocols have a strict request/response protocol. The memory operations on the microgrid, in contrast, are implemented as split-phase operations. Normally, the memory request itself carries the continuation for the writeback. For I/O operations, we assume that the I/O protocol cannot be altered, as this would break compatibility with most, if not all, existing I/O protocols and hamper adoption of the microgrid architecture. Ideally, of course, future I/O protocols would be designed to handle multiple outstanding requests. Until then, the I/O controller will be re-

---

1 For simplicity, the term I/O bus is used even though any form of I/O interconnect may be used (e.g., packet-switched network, daisy-chained link, etc.)
Chapter 5. The Chip Architecture

<table>
<thead>
<tr>
<th>Component</th>
<th>Area $mm^2$</th>
<th>%</th>
</tr>
</thead>
<tbody>
<tr>
<td>Cores</td>
<td>72.65</td>
<td>76%</td>
</tr>
<tr>
<td>L2 Caches</td>
<td>12.30</td>
<td>13%</td>
</tr>
<tr>
<td>FPUs</td>
<td>5.71</td>
<td>6%</td>
</tr>
<tr>
<td>Root Directories</td>
<td>3.81</td>
<td>4%</td>
</tr>
<tr>
<td>Directories</td>
<td>0.94</td>
<td>1%</td>
</tr>
<tr>
<td><strong>Total</strong></td>
<td><strong>95.41</strong></td>
<td><strong>100%</strong></td>
</tr>
</tbody>
</table>

Table 5.2: Area breakdown of an entire microgrid.

required to store the state of the split-phase I/O operations and provide the interface between the microgrid and existing I/O bus protocols.

Interrupts—that is, events from the I/O devices that do not have a preceding request—are handled similarly as responses. If a thread wishes to “listen” to a certain interrupt channel, it can register one of its registers with the I/O controller and suspend on the register simply by reading it. When the interrupt occurs, the specified register is written, waking up the thread that read it. The thread can then dispatch a non-critical task that handles the interrupt via a detached create (see section 4.5.3) and start over, ready to accept another interrupt. A latch is provided in the I/O controller to avoid race conditions, where no thread has registered as a handler.

The details of this mechanism and of the I/O controller in particular were not relevant to the research in this thesis, but have been investigated by others[105, 78].

5.6 Size analysis

Section 4.13 analyzed the area and timing requirements of a single core. This section analyzes the required area and the breakdown into components of an entire microgrid. To this end, a system-on-chip was created by combining 128 of these cores with 64 FPUs and 4 MB of L2 cache spread across 32 4-way set associative caches. The caches are divided into 4 rings, which are connected to the top-level ring containing 4 root directories, each connected to a DDR3-1600 module. This system-on-chip has a combined off-chip peak memory bandwidth of 51.2 GB/s and a peak floating-point throughput of 64 GFLOPS. The area of this chip has been calculated in the same manner as in section 4.13 as 95.4 mm$^2$; the breakdown of this area is shown in table 5.2.

5.7 Summary

In this chapter and the one before it, we described the design and rationale behind the Microgrid, a scalable many-core system-on-chip using cores with support for hardware multithreading rather than non-scalable logic for increasing sequential performance. We described the mapping of the general microthreading concepts from chapter onto hardware instructions and synchronizing registers and showed how these techniques,
when combined, result in a multithreaded pipeline that does not require speculation and can avoid stalling with cooperation from the compiler, at little to no extra cost.

Of course, a single core is nothing without a greater whole to be placed in, so in this chapter we described the composition of many microthreaded cores into a microgrid. In section 5.1 we described our design of the general composition of cores into a System-on-Chip with memory and network connections. By choosing a particular layouting algorithm, we made sure the design is scalable in a manner that is transparent to the design of a single core.

In section 5.2 we describe how to expose the available cores on the chip to the software via a concept dubbed *places*. It shows an encoding scheme that we co-designed with the compiler research that allows the software to efficiently manipulate the distribution of its work across a section of a chip, without expensive system calls or knowledge about the total size of the chip. This scheme allows software to be backward and future compatible, by being agnostic about the particular size of the hardware its running on.

In section 5.4 we described an experimental memory architecture based on a cache diffusion memory architecture which aims to provide memory locality in a distributed system and in section 5.5 we described a general mechanism for connecting the microgrid to the outside world via I/O in a distributed and scalable manner that ties in with the concepts of microthreading.

Finally, in section 5.6 we analyzed the overall size of the components on the chip for a typical configuration.

The System-on-chip architecture presented in this chapter is directly reflected in publications P.1, P.4, P.6, P.7, P.8, P.11 and P.12.

Combined, this and the previous chapter answer the research question of whether the architecture is feasible and show what components are necessary. The experimental validation in chapter 7 will show that the design is viable as well.
Chapter 6

The MicroGrid Simulator

“Who cares how it works, just as long as it gives the right answer.”

- Jeff Scholnik

To evaluate the hardware model described in chapter 4, it has been implemented in a cycle-accurate software simulator written in C++ called the MicroGrid Simulator, or MGSim for short. MGSim was originally created to model and evaluate the behavior of processors consisting of many D-RISC cores. It has since been extended, remodeled and refined into a general platform for many-core chip development and simulation, and for the Micongrid specifically; it has been designed from the ground up to allow for a detailed study of the hardware/software interactions on many-core platforms implementing the previously explained hardware model.

Since its development, the simulator described in this chapter has not just been used to carry out the research described in this thesis. It is a piece of software which stands at the core of the research group. It has been used as the core research tool in five completed Ph.D. projects regarding compilation, operating systems, programmability, reliability and real-time aspects of the Micongrid (see section 8.2). In addition to a core tool in the research group, it is also being used as a teaching tool in undergraduate and graduate computer architecture courses (see section 8.2).

The design and functioning of the simulator and its components as described in this chapter is entirely—except for the asynchronous monitoring system (see section 6.6.2)—our original contribution: what started as a “simple” reference implementation of the architecture in chapters 4 and 5 grew into the generic, modular System-on-chip simulator that is presented here.

6.1 Emulation vs. Simulation

MGSim is called a simulator, but historically the terms emulation and simulation are often used interchangeably. Practically, MGSim can be seen both as a simulator and an emulator. Simulators reproduce a component’s behavior at a high level, often to make predictions about them. They are key items for hardware architects, who need to explore the behavior of computing components before their designs are finalized.
Emulators, on the other hand, reproduce the concrete behavior of a component and are used, for example, during compiler and operating system development, to debug software in a controlled environment. Emulators can be further divided between partial and full system emulation (also called “virtualization”). With the former, only application code runs within the emulation environment, and operating system functions are serviced through a host/guest interface. With full emulation, the entire software stack runs on the emulated hardware. MGSim can serve both as a hardware simulator and full-system emulator. These differences are illustrated in figure 6.1.

As illustrated in figure 6.1, MGSim is part of a collection of frameworks for hardware simulation and/or emulation. It is most closely related to SimpleScalar [107] and M5 [108], so it is fair to ask why neither of the two were chosen as an existing platform instead of developing one from scratch.

SimpleScalar only provides partial emulation: operating system functions are served via the syscall pseudo-instruction. This makes SimpleScalar insufficient to study operating software, an aspect that is important to the Microgrid since the architecture is advocating stronger ties between hardware and software in the area of concurrency management. Furthermore, SimpleScalar is geared towards studying single-core platforms and not multi-core platforms, like MGSim.

The gem5 simulator system, or M5 for short, is a “modular platform for computer system architecture research, encompassing system-level architecture as well as processor microarchitecture”. While initially seeming like a good match, it is aimed for architecture research on traditional systems. It comes with operating systems and boot loaders for classic architectures. Since the architecture described in this thesis steers away from traditional architectures, it was felt that using an existing platform like M5 would complicate matters unnecessarily.

Other popular hardware simulation frameworks are SystemC [109] and VHDL [110] & Verilog [111]. SystemC is a widely-known IEEE-ratified language implemented by extending C++ with class libraries. It is used to model and evaluate system designs. However, the default SystemC implementation is notoriously slow and is too low-level for being able to quickly iterate over various architectural models. Moreover,
the licensing system of SystemC is relatively restrictive, which would prevent the wide
distribution, evaluation and reuse of MGSim by the community of peers. VHDL and
Verilog are two well-known hardware description languages that allow for prototyping
and simulating of computer hardware systems. The benefit of these language is that
a system described in VHDL or Verilog can be easily synthesized to, for instance,
an FPGA system. This then allows for evaluation of the design on actual hardware.
However, the level of description in these languages lies at the level of signals and
ports; too low for fast simulation of relatively large programs. Moreover, because
of the low level of description, designing an entire system becomes a tedious and
time-costly effort.

After considering all these options, it was decided to implement a simple but
fast cycle-based event scheduler supporting many-core platforms from scratch and
build the architectural model on top of that. Most work would be in the area of
architecture-specific implementation, so reusing any framework would only allow for
marginal gains.

6.2 Development principles

Given the central role of the simulator in the research conducted in this thesis as
well as any future research, special care had to be taken to guarantee correctness
and repeatability of the simulation. Without either, the results that are obtained from the
simulator are, essentially, useless. In addition, to support future development of and
enhancements to the simulator for future research, simplicity of implementation of
was another important factor in its design. And going without saying, the simulator
should run as fast as possible.

To guarantee these properties, several principles were adhered to during develop-
ment:

- The programming language of choice was C++. Wanting a simulator that runs
  as a fast as possible, limited the choice of language to non-interpreted and
  non-byte code languages such as Java or C#. With C and C++ the primary
  remaining choices, C++ was chosen for support object-oriented programming,
  which would simplify the code. In addition, the language was very familiar to
  the researchers, which sped things up as well.

- Threads should be avoided for the main simulation. Parallelizing a simulation
  with many communicating components will be difficult to begin with, and the
downsides that come with multithreaded programming outweigh the possible
gains. Performance increase can be obtained by running multiple instances of
the simulator in parallel, each working on a different problem scenario.

- The code should be simple and clear. By writing simple code, it is easier to
  understand what the code does, thus aiding in the verification process of the
  simulator. Although this might impact performance, this can be mitigated
  by running several instances in parallel. Also, there is some truth to the old
  adage that “premature optimization is the root of all evil”. The benefit of
having confidence in the result of the simulation is worth the expected loss in performance. Practically, this principle meant that the architecture specification should be separate from the simulation framework, and that the former should be written with modular components with clear and intuitive behaviors that map (nearly) one on one to hardware components.

6.3 Design

The simulator is designed as a collection of interconnected components that “implement” the simulated architecture. These components define hardware processes, buffers, queues, buses and memories. All of this is driven by the simulation kernel which schedules processes and storage updates according to their associated clock frequencies. In other words, the kernel is responsible for running the simulation. The components are created by the configuration logic, which typically reads a configuration file in order to determine which components to instantiate. The configuration logic is thus responsible for the creation of components and their cleanup once the simulation has finished. Some components in the simulator are, for instance, register files, pipeline stages, caches, memory controllers, thread managers and on-chip network controllers. These describe the logical division of a microgrid of microprocessors.

The kernel and the configuration are both created from the front-end, which acts as the user interface to the simulated system. Currently, it is implemented as a command-line program where the user can enter commands to advance the simulation, inspect components, set breakpoints and so on. Finally, the front-end can also instantiate a monitor. The monitor asynchronously monitors the simulated components to gather statistics. It can, for instance, monitor for how much time components are busy, how full buffers are, or how much contention there is on buses. These measurements are then dumped to files, where a python script aggregates and selects the data based on the user’s interest and can output the data in a format suitable for GNU plot for instance. An overview of this architecture is shown in figure 6.2.

6.3.1 Processes, Buffers and Arbitrators

The simulation kernel deals primarily with the scheduling and execution of processes. Processes represent stateless functional circuits in hardware that only access other stateful resources at points explicitly identified in their simulation model and visible to the simulation kernel; they are implemented as functions which are declared to be sensitive to one or more buffers. Buffers are FIFO storage components which can have as most one process be sensitive on them. When a buffer is written to, its sensitive process is scheduled for execution. Buffers can have multiple writers. When a process is run, it typically consumes some data from an input buffer and produces output data immediately, or at some point in the future. As mentioned, buffers can have multiple writer processes. However, typically only one process can write to a FIFO queue at a time, so an arbitrator must be used in conjunction with a buffer. Arbitrators are utility objects that determine access to shared resources (e.g. a buffer) according to some algorithm (e.g., static priority, or round-robin priority).
Figure 6.2: Overview of MGSim’s architecture.
The simulation terminates successfully when no more processes are scheduled to run. This condition is inferred from the state of the buffers that the processes are sensitive on. Since a microthreaded processor is a data-driven multithreaded processor, it is possible for all threads to terminate in which case the processor has no more work to do. This is different from traditional, sequential processors, which keep executing instructions until issued some sort of HALT instruction.

6.3.2 Clocks

In the simulation, clocks are utility objects that represent a clock signal with a certain frequency. Every active process or updatable storage element in the simulation is attached to a clock. This mechanism allows storage elements and processes to be placed in certain clock domains, or operate on other frequencies. These frequencies are defined in the simulation configuration, thus enabling quick testing of the architecture with different frequencies for the cores, external memory, memory network, or inter-core network, to name but a few.

Note that the ability to support multiple clock domains is especially important in multi-core simulations. The Intel SCC[34], for instance, can have different clock domains for processing cores and the on-chip network. Add in load-based frequency scaling and it becomes obvious that a proper simulation of modern multi-core architecture has to deal with different clock domains.

The simulation is implemented as an event-based simulation where, logically, a master clock is advanced tick by tick, executing processes and updating buffers as need be. The master clock is a frequency that is derived from every clock frequency that has been defined in the configuration by finding the least common multiple of all clock frequencies. Thus, every defined clock is guaranteed to have its tick on an integer multiple of a master clock tick. For instance, a configuration which defines a 1 GHz clock and a 800 MHz clock will run with a master frequency of 4 GHz. The 1 GHz clock will have a tick every 4 ticks of the master clock and the 800 MHz clock will have a tick every 5 ticks of the master clock. A linked list of “active” processes and storage elements is maintained per frequency domain, so the main simulation loop is an event-based loop, where the master clock tick count is simply advanced at once to the next tick where any process should be run.

6.3.3 Clock phases

Unlike actual hardware, the simulation is sequential in nature. Processes are run in a certain (in principle arbitrary) order, and within a process, due to the software engineering requirement that components are well encapsulated, different accesses to structures or ports can be made, even without a process knowing exactly which. For instance, a process accessing a register file need not know which (arbitrated) port is used to access it. As a result of the software’s sequential nature and encapsulation, two problems arise:

1. How to implement arbitration on shared resources and avoid deadlock.
2. How to undo earlier state changes, if arbitration failed later on.
Arbitration requires every process to identify their desire to access the resource before the arbiter decides. In hardware, this is simple because all processes assert a signal at the rising edge of the clock, and only one gets notified that it has access at the falling edge of the clock. In a sequential system, this is a problem.

Furthermore, if a multi-part process requires arbitration in one or more of its parts, it must be able to roll back its changes in earlier executed parts if arbitration fails in later parts. For instance, in our modified pipeline, the Writeback stage can write to the register file and to the network controller. In theory, either access might fail due to arbitration on ports to storage. Yet, both accesses are completely encapsulated in independent components, that need not know of each other’s existence. To provide roll backs for arbitration operations across levels of encapsulation would be an error-prone approach.

The solution is to introduce three clock phases: the acquire phase, the verify phase and the commit phase. Each process is first called in the acquire phase, then in the verify phase and finally in the commit phase. This allows a process to first acquire the access to storage elements or services that it requires, then, after all processes have registered their intent, the arbitrators are run to decide the “winners”. In the verify phase, each process can check whether it has “won” all of its registrations and finally, in the commit phase, do the work.

To avoid code duplication and improve correctness, the same code is run in every phase, with conditionals in the code causing different behavior in the process: segments of a process’ code can be placed in a so-called “COMMIT” block, which is only executed if the process is run in the commit phase. The idea behind this technique is that the code can be written without worrying about global order. In the previous example, the Writeback stage can perform both parts, without worrying about the order in which they are executed (assuming there is no dependency). The way this works is as follows: if a process wants access to a shared structure via a port, it must first call a certain method on a port to acquire it. This method performs differently based on the current phase. In the acquire phase, the calling process is simply put on a list of processes that want to access the port. After the acquire phase is run for all active processes, all active arbitrators are run to decide which processes get to access which shared structures. Then, the verify phase is run for all processes. This phase executes the same, except that the acquire methods now verify if the process has been granted access to the structure and returns the result of this check. The process can be written such that if the acquire method returns false, it simply indicates a stall condition to the kernel. If all acquire methods return true, that means that the process has been granted access to every shared structure that it wants to access. In this case, the kernel calls the method again, but this time in the commit phase. Here, the acquire methods do nothing, but all the code in the “COMMIT” blocks is executed, which actually perform the state changes to the various structures that the code wanted access to.

When used consistently, this mechanism allows hardware processes to be described clearly and concisely without worrying about ordering or roll backs. A clear relationship becomes obvious between inputs and outputs. The resulting overall simulation behavior mimics the semantics of Transaction-Level Modeling (TLM) found in SystemC, albeit with a much leaner and thus more efficient implementation.
6.4 Correctness

In order to trust the software, its correctness must somehow be proven or guaranteed. Generally, there are considered to be two aspects to correctness: Verification (does it do what it’s supposed to do?) and validation (does it do what we want it to do?).

For validation, it is required to have the specification of what needed to be built, which is, loosely put: “a cycle-accurate microgrid simulation that could be implemented in hardware using current or near-future technology”. This was the only specification that drove the project and also provides the specification for the simulator’s verification. All taken together, there are roughly two parts to the verification and validation of the simulator.

6.4.1 Functional correctness

Does the simulator produce the correct output? This is the most important part of the simulator and ties in with its simulation aspect, as described above. If a program is run on the simulator, it must produce the correct output or else the rest of the simulator does not matter. This validation was done by compiling a representative selection of test programs for both the microgrid architecture as well as the native system that the simulator runs on and comparing their outputs. Provided that the program’s behavior itself is fully specified, this was a simple baseline test for the simulator.

Note that this dual target compilation is assisted by the fact that, as explained in section 3.4 microthreaded programs, can be automatically sequentialized due to the restrictions on the intra-family communication channels. Thus, this verification is as simple as writing the code once, and running tools to compile and run for both the simulator, and a native, sequential system. This feature of the language is also explained in more depth [77, 78].

6.4.2 Behavioral correctness

Does the simulator run correctly? This is the part that matters to the research and ties in with its emulation aspect, as described above. The program, even if it produces the correct output, must run on the simulator as expected according to the specified hardware architecture. This is hard to verify with automated tools, so the approach was to implement certain mechanisms in the core of the simulation that would force any implementation to be written in similar ways as the hardware would be designed:

**Single reader** Buffers cannot be read by more than one process. This makes buffers act as the input queues for hardware processes which have simple tasks. It also forces the model designer to choose an approach where resulting models can be abstracted as Kahn Process Networks, which can be analyzed for deadlock-freedom. Processes have a clearly defined behavior and responsibility of moving data from one buffer or storage element to another buffer or storage element, cycle by cycle.
**Delayed visibility**  When a buffer is written by a process, this write does not become visible to its consumer process until the next cycle. This has a double benefit. Primarily, it avoids the problem where the writing process could run before the reading process in the same cycle, thus propagating the value faster than would be possible in hardware. Secondly, it simulates the latency inherent to hardware processes.

The techniques above have been encased in the fundamental simulation architecture, and by strictly adhering to them the system as a whole can be guaranteed to be reasonably cycle-accurate and behave in an overall way that is implementable in hardware. Naturally, a hardware component could still be written that computes some extremely complicated result in a single cycle, but this is local behavior and could be easily spotted by code review.

### 6.5 Stalls and deadlock

Because of arbitration and bounded inter-process buffers, a process may be denied access to a shared resource in some cycle. Either another process has priority, or the buffer it wants to write to is full. In these cases, the process has to wait; it has to **stall**. This is indicated by a special return value to the simulation kernel from the process’ function.

If every runnable process returns a stall indication, the system is considered **deadlocked**. This can happen in practice because the inter-process buffers are of finite size. If there is a cyclical dependency between two or more processes, where each process is wanting to write to a buffer which is full and read by another process wanting to do the same, the processes are deadlocked. Such cyclical dependencies are common design errors, often found during early iterations of the design of new networked systems, and were encountered during the work described in this thesis.

To assist in debugging and analyzing deadlock scenarios and potentially proving deadlock freedom of the implemented hardware model, each process is extended with a set of approved storage **traces**. A storage trace is an ordered list of storage writes that a process could perform; the simulator can verify at runtime that a process is only writing to storages it is allowed to write to according to its storage trace. These traces enable the generation of a graph of all storage and process dependencies in the system. From this graph, any cycles can be identified and manually analyzed. Deadlock can be provably avoided if there are no cycles in the graph that are formed via buffers that could not contain all possible inputs.

It is conceptually possible to fully automate the formal checking of the process graph, however it was chosen to not do so in this thesis as we are exploiting specific optimization avenues that would be difficult to handle properly in state-of-the-art checkers. For example, although there is a cycle in the form of pipeline - scheduler - pipeline, there is no deadlock here, because the buffer used by the scheduler is by definition large enough to contain all possible threads in the processor, since the number of threads that can run on a processor is limited.

**Memory**  To simplify deadlock analysis, the memory network has been separated from the rest of the architecture. The memory, as described in section 5.4, is a cache-
based hierarchical ring network. In [101] it was proved that a unidirectional ring-based message-passing network can be guaranteed deadlock free if every node had at least two entries in its ring buffer and would only accept a message from outside the ring if it had at least two entries free in its buffer. This rule guarantees that every node will always have a buffer entry available for accepting a message from its previous node, thus ensuring that the messages on the network would always keep moving. In this memory, however, this matter is complicated due to the hierarchical nature of the network. The solution to this is to consider the flattened network as the main ring network on which deadlock freedom can be guaranteed. Messages forwarded across directories in effect shortcut across the ring. With respect to deadlock freedom, this shortcut should not be treated as a forward, since it could deadlock the ring it is shortcutting across could deadlock itself. The solution is to try to forward in directories if there are at least 2 entries available in the output buffer of the directory, and otherwise route down into the ring to follow the flattened ring.

6.6 Monitoring

So far, the simulator has been described in terms of features and programming interfaces that relate to simulating a hardware architecture. An important aspect that has been glossed over so far is monitoring: how to observe what is going on in the simulation and get meaningful results from the simulation that can be used in architecture research. To this end, the simulator contains three mechanisms to monitor and debug the simulation: synchronous event traces, asynchronous monitoring and performance counters. The first two exist to analyze, debug and monitor the simulated hardware and are invisible to the simulated program. The last one is an ISA-extension that allows the software run on the simulator to monitor certain performance-related variables during execution. All three mechanisms are explained in detail below.

6.6.1 Synchronous event traces

Synchronous event traces report all simulation events related to a specific trace category and is intended for troubleshooting or inspecting the detailed cycle-to-cycle behavior of components.

Every component or service in the simulator’s hardware model can use a simulator-provided API consisting of printf-like macros to report simulation events of a specific category. The user of the simulator can specify which categories should be output and which ones should be ignored. Trace categories which are ignored have minimal overhead in the simulation, because the tracing macros expand to code that only formats and prints the message if the corresponding trace category is enabled. A list of all defined trace categories is shown in table 6.1.

Every trace command outputs a line containing:

1. the simulation’s master cycle counter.

2. the name of the component that issued the trace event.

3. the name of the process that issued the event.
Chapter 6. The MicroGrid Simulator

deadlock  Events reporting a process stall
flow     Branch instructions and thread creation
fpu      Events in the floating-point unit
io       I/O operations
iot      Messages on the I/O network
mem      Memory load and store instructions
net      Events on the delegation/link network
pipe     Events reporting pipeline activity
reg      Accesses to a core’s register file
sim      General simulation events (default)

Table 6.1: Synchronous event trace categories

4. the event category (identified by a single character).
5. the text of the event.

An example trace for category flow is given in listing 6.1, illustrating the run-
time call tree of the program and how much time is spent in each function. This type
of tracing is essential for debugging the designed hardware components, especially
combined with running the simulator in interactive mode, where the simulation can
be stepped through one cycle at a time and introspected using the command-line.

Visualization  A set of utility tools have been developed to aid in analyzing the
behavior of the system via these event traces[112]. One important tool is the visual-
ization utility viewlog. Figure 6.3 shows example visualizations of the event traces
of running several computational kernels on a cluster of cores in the microgrid. The
visualizations indicate at what time certain processes are active. Horizontally, dif-
f erent processes of interest are listed next to each other, such as the D-Cache, the
pipeline stages of the cores and the FPUs that are shared among the cores. The ver-
tical axis represents time, in this case clock cycles. A square at location (x, y) means
that process x was active at cycle y. Squares in the pipeline stages are color-coded
to represent a unique thread context per color.

This type of visualization allows the hardware designer to quickly observe when
and how cores are active or allow him to identify bottlenecks in performance.
Chapter 6. The MicroGrid Simulator

Listing 6.1: Trace for a “Hello world” program.

6.6.2 Asynchronous monitoring

Asynchronous monitoring captures the changes of specific variables from the simulation model at regular time intervals. It is faster than synchronous event tracing described in the previous paragraph, but less accurate. It is meant for collecting statistical information over large time scales. The architecture of asynchronous monitoring is described in figure 6.4: a monitor thread runs concurrently with the simulation and repeatedly samples a set of selected variables at a configurable sample rate. The collected samples can be written to a binary file, or piped through an external utility. By default, the simulator comes with a utility called readtrace which can transform the binary samples to a textual format, suitable for plotting using e.g. GNUPlot. It can also be used to reduce and aggregate the monitor samples using a custom synthesis engine written in Python.

Figure 6.5 shows an example of an asynchronous trace: The classical Whetstone benchmark\(^1\) was run on a 16-core microgrid simulation using an Apple MacBook Pro as simulation host. The monitor thread was configured to sample the simulation cycle counter and the counter for the number of instructions executed in the cores’ pipeline at a target rate of 1000 samples/s. The simulation executed 763 thousand instructions in 4.22 seconds, with just over 2.3 milliseconds of simulated time. While the simulation ran, the monitor thread recorded 3763 samples for an effective sample rate of 891 samples/s. Figure 6.5 was produced using GNUplot after piping the recorded samples through the readtrace utility.

Aside from showing example output of asynchronous monitoring, figure 6.5 also gives an indication of the performance of the simulation itself. For the configured 16-core microgrid, the simulator ran at an average 181k instructions per real second. The figure shows that although this average speed is maintained during most of the simulation, it does fluctuate during certain parts of the simulation where more activity is going on and more components of the simulated cores are active than others.

The figure also shows that the simulation proceeds at roughly 600k simulated cycles per real second. From this we can compute a simulated IPC of approximately 0.3. This relatively “low” IPC is not surprising given the fact that the benchmark program is a sequential program run on a single non-superscalar core and thus benefits little from the parallelization facilities of the microgrid.

\(^1\)http://www.netlib.org/benchmark/whetstone.c
Figure 6.3: Example visualizations of synchronous event traces. A tooltip shows the details for the square that the mouse hovered over.
Chapter 6. The MicroGrid Simulator

Figure 6.4: Architecture of asynchronous monitoring

Figure 6.5: Performance visualization of the Whetstone benchmark and simulator
Chapter 6. The MicroGrid Simulator

### Component Files LoC

<table>
<thead>
<tr>
<th>Component</th>
<th>Files</th>
<th>LoC</th>
</tr>
</thead>
<tbody>
<tr>
<td>Simulation kernel</td>
<td>32</td>
<td>5097</td>
</tr>
<tr>
<td>Processor</td>
<td>57</td>
<td>18940</td>
</tr>
<tr>
<td>Memory: COMA</td>
<td>11</td>
<td>3684</td>
</tr>
<tr>
<td>Memory: ZLCOMA</td>
<td>10</td>
<td>3344</td>
</tr>
<tr>
<td>Memory: Others</td>
<td>10</td>
<td>2284</td>
</tr>
<tr>
<td>Architectural Support (FPU, etc.)</td>
<td>18</td>
<td>4817</td>
</tr>
<tr>
<td>I/O Devices</td>
<td>26</td>
<td>6272</td>
</tr>
<tr>
<td>Command-line Interface</td>
<td>17</td>
<td>1558</td>
</tr>
<tr>
<td>Monitoring tools</td>
<td>5</td>
<td>921</td>
</tr>
<tr>
<td>Tests Programs</td>
<td>97</td>
<td>8743</td>
</tr>
<tr>
<td><strong>Total</strong></td>
<td><strong>283</strong></td>
<td><strong>55660</strong></td>
</tr>
</tbody>
</table>

Table 6.2: Simulator statistics

#### 6.6.3 Performance counters

Both monitoring mechanisms mentioned so far are “external” to the simulation, i.e., they are invisible to the simulated program. The simulator also provides an ISA-level interface that allows the program to inspect predefined performance-related counters. Some of these may already be standard in the base ISA. For instance, the Alpha ISA has the \texttt{rpc} instruction that reads the core’s current cycle counter. The simulator also exposes other counters, such as number of execution instructions, number of issued memory requests, and so on. These counters are exposed via the memory interface. Loads from reserved address ranges are intercepted by the simulator and immediately return the counter’s value.

This mechanism has been implemented to enable precise reporting of simulator statistics between start and end points of a computation which have been specified in the program’s source code itself. It is specifically not intended as a realistic model of performance counter interfaces found in real hardware.

#### 6.7 Statistics

To round up this chapter, we present the current state of the simulator in table 6.2. These statistics have been based on the \texttt{mgsim} repository, at the time of writing commit 61949baa94e35b8cede29c9df436ccab47fc038a, dated 11 October 2013.\[2]\[2\] The current codebase is just shy of 60,000 lines of code spread out over just under 300 files. The single largest component is the Processor simulation, at roughly 19,000 lines of code in 57 files.

\[2]\[2\]The \texttt{mgsim} repository can be found at https://github.com/svp-dev/mgsim/tree/master/
Chapter 6. The MicroGrid Simulator

6.8 Discussion

It is worth noting that MGSim is similar to Gem5 [108], another C++-based framework for event and component-based multi-core simulation. Both framework have comparable performance, simulating processors models with thousands of components at 100-1000KIPS on contemporary desktop-grade hardware. However, where Gem5 focuses on compatibility with real hardware and accuracy for models with few cores, MGSim focuses on simplicity and comprehension of implementation and accuracy for models with many cores.

Future work for the simulator exists to address the two largest problems in the current implementation. Firstly, it is extremely common to introduce design errors when developing the hardware model. Such errors often lead to deadlock in ways that are not easily predicted beforehand. Although the underlying simulation framework exposes all relationship between buffers and processors, there exist no tools yet to perform deadlock analysis on the implemented model. Implementing such an analysis in the simulator would significantly reduce the time spent debugging the hardware model. Secondly, the simulator has no checkpointing facilities. If an error occurs a long time into a simulation, the only way to reproduce it is to run the simulation again from the start. However, if the program depends on external non-deterministic input via the I/O subsystem, an exact re-play might not be possible. Therefore, a valuable addition to the simulator would be the possibility to “checkpoint” the state of the simulator at specified or regular intervals, thus allowing for a quick and easy replay of the simulation up to several cycles before the point where the error occurred.

Although the simulator was primarily developed as a result of this research, it is still being used and developed at the University of Amsterdam and its partners for ongoing research on the microgrid architecture[113] and graduate-level education on processor, cache and memory architectures. Since its inception and initial development, several members of the research group have extended the simulator with some of the features mentioned in this chapter. This effort continues to date, re-establishing the fact that MGSim is a core tool in current and future architecture research at the University of Amsterdam.

6.9 Summary

This chapter presented another contribution of this thesis next to the architectural work and design: MGSim, a cycle-accurate simulator written in C++ that was developed in order to evaluate the architecture described in chapter 4. It consists of a reusable event-driven, highly configurable, multi-clock domain hardware simulation platform and an architecture-specific simulation that runs on top. The latter consists of a library of components that implement a hardware multithreaded in-order RISC core, several different memory systems and an I/O subsystem in order to allow for full system emulation. It contains comprehensive inspection and monitoring facilities that make it suitable for both architecture research and education.

The existance of MGSim, along with the implementation of the architectural design in it, answers the first part of the main research question from section 1.4: is the
Chapter 6. The MicroGrid Simulator

architecture feasible? After presenting this chapter, it is our opinion that this question can be answered positively by virtue of the design existing in an implementation that was designed with realistic constraints at its core.

The simulator presented in this chapter is also presented in publications P.1, P.2, P.3, P.4, P.5, P.6, P.7, P.8, P.10, P.11 and P.12.

The next chapter will use the simulator described in this chapter to obtain experimental results of the System-on-Chip architecture described in chapters 4 and 5 and thus answer the second and last part of the main research question: is the architecture viable?
Chapter 7

Evaluation

“There are three kinds of lies: lies, damned lies, and statistics.”
- Mark Twain

Although a significant amount of evaluation was essential in the development process described in chapters 4 and 5, this chapter presents an in depth investigation and analysis of the challenges outlined in the introduction. In order to achieve this, we ran (and, if not yet available, developed) several representative tests and benchmarks using the simulator described in chapter 6. As indicated in section 7.1, the high-level programming tools for these benchmarks were made available from associated programming language and compiler research within the research group; the rest of this chapter (the writing, running and evaluating the presented benchmarks) is our original work, and has been presented in a variety of joint publications from members of our research group.

7.1 Programming Tools

In order to program the Microgrid, a new language was developed: SL [77]. This language was developed out of a need for being able to write microthreaded code while lacking a proper compiler for the Microgrid. SL was pragmatically developed on top of existing technologies to create a microthreaded-capable toolchain capable of targeting multiple platforms in the shortest time possible. The compiler of SL, slc, is capable of compiling C files with microthread extensions. It has been implemented as a set of code transformation rules written in M4, using inline assembly for the Microgrid target and runs on top of the GNU C Compiler using a version of the GNU Assembler that has been extended with the microgrid ISA extensions.

SL is capable of targeting different platforms. Depending on the flags passed to slc, it can produce a binary for the Microgrid, where the microthreaded constructs are mapped relatively straight forward to the microthread instructions in the extended ISA, or to a native, sequential binary, where families and threads are transformed into loops. This allows benchmarks to be written once in SL, compiled and tested natively,
and then benchmarked on the microgrid. Being able to implement the benchmark natively significantly reduces the time between iterations during development.

Additionally, SL comes with a cross-target library implementing standard methods such as string formatting, time methods, profiling and I/O. On the microgrid, the implementation of this library ties in with the mechanisms described in the previous chapters.

For a detailed explanation of this toolchain and its technologies, see [77].

7.2 Configuration

Unless otherwise specified, the architecture was simulated with a set of default parameters, which are as follows. The clock frequency of the cores and memory network was set at 1 GHz. The L1 I-Cache and D-Cache were 4-way set associative with 64-bytes lines. Both are relatively small, with the I-Cache storing 2 kB and the D-Cache storing 4 kB. Each core had space for 32 family entries and 256 thread entries. The integer and floating-point register files were configured at 1024 registers each, for a ratio of 4 registers per thread.

The FPUs were modeled with a latency of 8 and 10 cycles for division and square root respectively and 3 cycles for addition, subtraction and multiplication. Each FPU is shared between two cores and has an input queue for each core with two-way issue onto 5 execution units: 2 identical units for addition and subtraction and 3 dedicated units for multiplication, division and square root, respectively.

The L2 COMA cache is a 128 kB 4-way set associative cache shared among 4 cores via a bus. 8 COMA caches are connected via a daisy-chained ring network per COMA directory. The external memory was configured to simulate DDR-1600 memory.

7.3 Numerical kernels

The first and simplest way to evaluate the architecture was to run several small and different numerical kernels on it. This evaluation shows the fundamental way in which the Microgrid offers performance improvements: increased efficiency by interleaving of multiple threads on a single core, and increased throughput by distributing multiple threads across multiple cores. This is illustrated in figure 7.1. This figure shows that as the number of threads per core goes up, pipeline utilization goes up to nearly full utilization at 8 threads/core for computation-intensive kernels. The naive implementation of Quicksort does not benefit from hardware multithreading due to its sequential pivot-selection algorithm and performs poorly overall due to an absence of branch prediction in hardware. In comparison, Mergesort performs better because the natural concurrency in even a naive implementation allows it to benefit from multithreading.

Figure 7.2 shows the scalability of different kernels relative to their single-core, single-threaded execution. This shows that distributing the algorithm across a few cores leads to improved performance. In particular, the figure highlights that it is more advantageous to distribute a number of threads across cores than interleave
Chapter 7. Evaluation

Figure 7.1: Average IPC per core for several numerical kernels ran on a single core with 1–16 threads/core.

them on a single core. However, given that the number of cores is usually limited, interleaving multiple threads on each core does offer further benefits.

7.4 Reduction

Another interesting case study for the Microgrid is reduction. Conceptually a sequential operation, it can be parallelized in many ways. Here, three approaches are considered for computing the sum of an array. The first is to divide the work evenly across $P$ cores with one family per core, each calculating a partial sum. Then an overall family of $P$ threads collects the partial sums and adds them. The second approach, $N/CP$ is a variation on this approach where every core runs not 1, but $C$ reductions in parallel to benefit from thread interleaving. While $N$ and $P$ are given, $C$ can be chosen, but care should be taken. When too large, too much overhead is spent on creating families that are too small. When too little, not enough parallelism is created to fully utilize every core. To find the optimal value of $C$, we realize that despite the parallelization of work, there are still three sequential paths that can dominate the overall runtime, in accordance to Amdahl’s Law:

- A loop of $N/CP$ threads (the inner-most family that sums a subset of the array).
- A loop of $C$ threads (the family that sums the reductions from the previous families on a single core).
- A loop of $P$ threads (the family that sums the reductions across all the cores).

The overall sequential complexity of the $N/CP$ algorithm becomes $\max(N/CP, C, P)$, i.e., the largest component dominates the entire algorithm. Thus, we wish to solve:
Figure 7.2: Scalability of numeric kernels for 1–4 cores.
argmin max(N/CP, C, P). Since \( P \) is not affected by changes in \( C \), this can be simplified to argmin max(N/CP, C). Given that \( N/CP \) and \( C \) behave inversely to changes in \( C \), the only solution is to solve for the point where they are both equal: \( C = N/CP \). Which solves as: \( C = \sqrt{N/P} \).

The last approach is an adaption of the parallel prefix sum \[114\], where \( \log N \) families of \( N/2 \) threads down to 1 thread are created in turns to reduce the array in-memory down to a single element.

Figure 7.3a shows the baseline comparison between the \( N/P \) approach and a baseline sequential implementation running on an Intel P8600 at 2.4 GHz. The results show that the parallized version scales well with the number of cores used and matches the baseline performance from 16 cores onwards. This is especially noteworthy considering that the Intel P8600’s die is 107 \( mm^2 \)[115], which is roughly the same area as a Microgrid of 128 cores (95.4 \( mm^2 \)).

Figure 7.3b shows how the various approaches perform with respect to the size of the input array, when run on 64 cores. It shows that the \( N/P \) and prefix sum approach perform comparably for lower input sizes, but the \( N/P \) approach gradually starts to outperforms the prefix sum. This is not surprising given that the prefix sum has \( O(N \log N) \) operations and the \( N/P \) sum \( O(N) \). Both approaches are entirely outperformed by the \( N/CP \) approach, which gets its additional performance over the \( N/P \) approach from its increased core utilization. Given that each leaf thread is particularly small (i.e., an offset calculation, a memory load and an addition), this shows that enough on-core parallelism can effectively hide memory latency even for small threads.

### 7.5 Cryptography

Two hashes, two stream ciphers and six block ciphers in CBC mode were run on streams of data to evaluate the throughput of the system. Table 7.1 lists the kernels that were timed, and their parameters. The performance of these algorithms on
Chapter 7. Evaluation

<table>
<thead>
<tr>
<th>Name</th>
<th>Block Size</th>
<th>Rounds</th>
<th>Key Size</th>
<th>Table Size</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>(bits)</td>
<td>(bytes)</td>
<td>(bytes)</td>
<td>(bytes)</td>
</tr>
<tr>
<td>AES</td>
<td>128</td>
<td>10</td>
<td>16</td>
<td>8192</td>
</tr>
<tr>
<td>DES</td>
<td>64</td>
<td>16</td>
<td>8</td>
<td>4096</td>
</tr>
<tr>
<td>RC5</td>
<td>64</td>
<td>16</td>
<td>16</td>
<td>0</td>
</tr>
<tr>
<td>RC6</td>
<td>128</td>
<td>20</td>
<td>16</td>
<td>0</td>
</tr>
<tr>
<td>IDEA</td>
<td>64</td>
<td>9</td>
<td>16</td>
<td>0</td>
</tr>
<tr>
<td>Blowfish</td>
<td>64</td>
<td>16</td>
<td>56</td>
<td>4096</td>
</tr>
<tr>
<td>RC4</td>
<td>-</td>
<td>-</td>
<td>16</td>
<td>256</td>
</tr>
<tr>
<td>SEAL</td>
<td>-</td>
<td>-</td>
<td>20</td>
<td>&lt;4096</td>
</tr>
<tr>
<td>MD5</td>
<td>512</td>
<td>64</td>
<td>-</td>
<td>0</td>
</tr>
<tr>
<td>SHA-1</td>
<td>512</td>
<td>80</td>
<td>-</td>
<td>0</td>
</tr>
</tbody>
</table>

Table 7.1: The cryptography kernels and their parameters

A Microgrid of 1 GHz cores was compared with the performance for a functionally similar benchmark suite called NPCryptBench which ran on specialized network processors (Intel IXP chips). The best IXP chip that was evaluated, the Intel IXP 2800, has 16 cores (“MicroEngines”) running at 1.4 GHz, controlled by a single XScale processor for controlling the program and dispatching work to the MicroEngines. Additionally, it has hardware hash units to speed up common hash algorithms, a common operation in network code.

First, the throughput for NPCryptBench’s unoptimized code benchmark was compared against the throughput of the simply parallelized code on the Microgrid. These results are shown in figure 7.4. These are the results for a single data stream being encrypted or hashed on a single core (on the Microgrid) and MicroEngine (on the IXP chips). The results show the Microgrid outperforming the IXP chips on every cipher but the RC and hash ciphers, despite being clocked nearly 30% slower than the IXP MicroEngines. The RC ciphers have a carried dependency in a small loop that severely limits the exposed concurrency and practically serializes execution. In this case, the faster clocked IXP MicroEngine outperform the Microgrid core. For the hashing algorithms the IXP can easily beat the Microgrid due to the hardware hash units that it has, so this is no surprise. Of course, specialized hardware means the IXP chips are less suitable to be used for anything else, so they pay in generality.

In particular, SEAL is able to perform so much better than the rest because SEAL encrypts a stream in independent blocks. Unlike AES or DES which, in CBC mode, carry a dependency all the way through the encryption stream, SEAL only does this within blocks. The blocks themselves can be encrypted fully independently and in parallel. This feature allows the algorithm to perform extremely well on the Microgrid.

With this baseline comparison established, the next evaluation was one where multiple data streams are processed in parallel on the IXP and Microgrid for the most popular cryptographic kernels from these benchmarks. Here, several streams were processed in parallel on multiple cores and MicroEngines. In this comparison,
NPCryptBench was heavily optimized to take full advantage of the IXP’s architecture: it uses inline assembly, manually threading specific regions of rounds and tweaking the burst performance of the memory system. This code, which runs on the MicroEngines, has to be compiled separately from the control code around it. In contrast, all parts of the Microgrid code run on the same kind of cores and was written generically without inline assembly or splitting small statements into threads. The results of this comparison are shown in figure 7.5. Here we can again see that the IXP outperforms the Microgrid for the hashing functions, due to its hardware hash units. In all other cases, the slower clocked microgrid performs comparable or even better than the IXP as the amount of parallelism increases. And where we see the IXP’s results flatten out at 800 Mbps for most of the kernels that don’t use the hardware hash units, the Microgrid exceeds this.

This evaluation shows that, barring hardware optimizations, the generic Microgrid will outperform specialized processors as parallelism increases, even without laborious platform-specific code optimizations.

7.6 Summary & Discussion

This chapter presented an evaluation of the architecture described in chapter 4. Admittedly, there are far more benchmarks and tests that we would have like to run and investigated on the architecture, however this was not possible for purely practical reasons. To understand why, it must be understood that this research took place during the initial phases of the group’s research on this architecture. A compiler for the architecture did not exist until a few years into the research effort, which means that until then only trivial programs written in assembly (assembled with a modified GNU assembler for the Alpha instruction set) could be run on the architecture. Once larger programs could be compiled and run instead, deadlock issues
Figure 7.5: Throughput for 1, 2, 4, 8 streams per core on 1–16 cores (or MicroEngines). IXP2800 performance for 1, 4, 8 streams/core in 3 leftmost bars in each group.
and architectural design flaws popped up. Addressing these required additional time and effort before a stable architectural design was available for proper benchmark analysis.

Furthermore, the focus of the research presented in this thesis lay in the design and validation of the core architecture; this explicitly did not include the memory system, which was independently investigated by other members of the research group [97, 98, 119, 120]. But, as they say, the devil is in the details; integrating this memory system into the simulator with the actual architecture simulation identified edge cases, race conditions and memory consistency problems, which had to be solved by changing the memory system design. Only then were all the pieces in place and could testing with a realistic process architecture and memory system begin.

In the end, the research partners have carried out a more extensive evaluation than presented in this thesis, as reported in the follow-up literature. Although these results were not obtained by us and cannot be reported here as original, they further confirm the viability of the proposed architecture. Nonetheless the research achieved its goal in the development of a fully functional architectural design and simulation, including memory system. This design now serves as a platform for continued research within the group, investigating follow-up issues such as system software issues [121, 122, 123, 77], fault tolerance issues [124, 125] and algorithmic issues [126].

Despite these issues, the results that are presented here show that the architecture is capable of multithreaded and scalable execution, and in some benchmarks even shows comparable or improved performance with contemporary and application-optimized hardware, despite being a general processor.

In particular, the results show that the architecture has successfully met the challenges laid in section 1.4. The high pipeline utilization shown in section 7.3 indicates an efficient system: nearly full utilization with compute intensive kernels is reached with 8 threads/core; a fraction of the available thread entries. The fact that the design can be instantiated with any number of cores with only local connections, shows that the design is scalable. This is backed by the results which show scalable performance increase as the number of utilized cores increase. Furthermore, the binaries were compiled agnostic of the system configuration, which proves the forward and backward compatibility of the design.

The results presented in this chapter are also presented in publications P.1, P.2, P.3, P.4, P.5, P.6, P.7, P.8, P.10 and P.11 in the publications list at the end of this thesis.
Chapter 7. Evaluation
Chapter 8

Conclusions & Future Work

“The future will be better tomorrow.”

- Dan Quayle

This thesis presented a novel multithreaded multi-core architecture based on a programming model which allows crucial aspects of thread management to be implemented in hardware with the goal of being scalable and future-proof by supporting fine-grained concurrency in software while offering binary compatibility. This newly designed architecture can be implemented around any von Neumann-based instruction set and defines a set of hardware structures, connections and behaviors to support the Microthreading programming model in hardware. Running on this architecture gives software the ability to easily and quickly turn large amount of concurrency, both functional concurrency and data concurrency, into physical thread-based parallelism with low overheads for inter-thread communication within groups of concurrent activities.

The aim of the research was to discover if this kind of architecture would be feasible and viable despite the many challenges associated with it. To this end, chapters 4 and 5 presented the main contribution of this research: a von Neumann-dataflow hybrid architecture that uses multi-threading and worst-case thread switching hints from software to hide the latency of long-latency operations such as memory loads and floating point operations.

The design process was thoroughly energy-conscious, and purposely avoided techniques from contemporary architecture that are energy-wasteful such as speculation, busy waits, predicated execution, implicit parallelism extraction and large caches, replacing them with passive, high-utilization and high-efficiency structures as needed.

Additionally, the architecture is competitive in terms of performance per area with contemporary general microprocessors due to the fact that many specialized structures that deal with optimizations such as out-of-order execution, branch prediction and cache prefetching do not need to be implemented to achieve comparable performance. Instead, a lot of the reclaimed space is spent on increased register files, simple thread management structures and replication of cores in order to expand the available on-chip parallelism.

Chapter 6 presented the technical contribution of this research: a cycle-accurate software simulator that was used to simulate this architecture, and shows that the sim-
ulation is representative of a physical implementation due to its clear and structured composition, and its adherence to the same limitations as a hardware implementation would be.

Chapter 7 evaluated the execution of several applications and kernels on this architecture and shows that the designed architecture, simulated on the developed simulator, performs as desired and even rivals contemporary implementations in terms of efficiency for selected benchmarks. This positively answers the question of whether the architecture is viable.

To summarize, the contribution of the research and work described in the thesis consisted of:

- the requirements for the design of the architecture and the identification, analysis and solving of bottlenecks and other problems related to its implementation.
- the design and construction of the cycle-accurate software simulator and its use to validate and verify the design of the architecture.
- the implementation of the benchmarking programs in the Microthreading programming language and their use in the subsequent evaluation of the architecture.

In the end, through these contributions, this thesis has shown that the Microgrid architecture is indeed feasible and viable!

8.1 Discussion

The architecture described in this thesis has taken an approach to multithreading where the execution contexts of an unusually large number of threads are stored in hardware, enabled via an equally unusually large register file. This extra space is available because of sacrifices in cache space and hardware optimizations for sequential programs. A size analysis based on the implemented hardware structures shows that this implementation is not only feasible, but even requires less silicon area than contemporary processor architectures.

A major improvement of the Microgrid processor architecture over other architectures is the switch command interleaved in the instruction stream by the compiler. This construct allows the pipeline, given multiple threads, to avoid speculation by switching to another thread when the compiler indicates that a long-latency operation is present. By dedicating a bit in the instruction stream for this, and altering the I-Cache to allow for per-cache line lists of waiting threads, the pipeline can avoid waiting until the Decode stage before it knows whether the current instruction is a long-latency instruction. This simple construct alone allows the pipeline to continue at near 100% utilization without the addition of expensive hardware such as branch prediction. This construct is also different from earlier interleaved multithreading architectures, which always interleave every cycle. The latter has the downside that a processor needs as many threads as there are pipeline stages. Of course, the solution in the Microgrid processor only works as long as there is at least another thread available to execute. But, as we’ve discussed in section 2.5, a system can be designed
with a mix of Microgrid processors and processors optimized for sequential execution. Such a processor could still support the thread and family management facilities from this architecture, but sacrifice thread and family entries for hardware structures which optimize sequential execution. As long as the same network protocol is implemented, the system would be able to run programs on every processor and scheduling software would be responsible for ensuring that programs run on those processors that are best optimized for them.

The goal of the architecture was to be energy-efficient; the microthreaded core was designed to minimize or flat-out remove non-scalable and/or energy-inefficient techniques such as cache prefetching, branch prediction, predicated execution, out-of-order execution and pipeline stalls. Avoiding the latter gave rise to the split-phase long-latency operations with dataflow scheduling on registers and compiler-assisted 0-cycle thread switching. All these techniques were designed and tuned to avoid pipeline stalls. So although the energy benefits of this design could not be quantified in this research, we believe the benefit is there, and further research could quantify it and, where possible, improve it. As an example, the thread-driven nature of the cores allows them to be extended with logic for clock frequency scaling or even suspending of inactive cores. With many such cores on a single chip to the point where cores can be dedicated to applications, the chip could conceivably cooperate with the operating system and suspend entire sections of the chip that are not used.

Next to developing the architecture, this research has also produced a cycle-accurate simulator of said architecture. This has proven to be a practical and extremely valuable means to achieve experimental validation and is a first step to show and convince the research community that an implementation is feasible. To ensure this last point, the simulator has been developed according to strict guidelines which ensure that it behaves like real hardware and has to respect certain restrictions like real hardware. In addition, it also uses several techniques to ensure that the code implementing the architecture-specific simulation is easily understood and error-proof. Although these techniques might be detrimental to the overall performance of the simulator, they are invaluable when it comes to placing trust in the output of the simulator. This is why the results in this thesis can be presented with confidence.

From a strategic point of view, the simulator is at least as great a result of this research as the architecture itself. With a reusable simulator framework and a clear, high-level and easily maintainable architecture implementation in place, further research can be easily performed, significantly speeding up the research associated with this architecture. As the next section shows, this research has already begun.

### 8.2 Impact

The research described in this thesis has laid the foundation for a continued research and teaching effort. In particular, it has spawned or supported:

- five successful Ph.D. projects using the simulator and the Microgrid architecture \[76, 77, 78, 79, 80\].
- four B.Sc. and one M.Sc. graduation projects, all with high grades.
• a well-rated assignment at a Computer Architecture course at the University of Leiden.

• three collaborations with international researchers.

• an industrial cooperation with the European Space Agency, for the development of a prototype.

This research and the developed architecture and simulator have sat at the core of three successful funded projects: Microgrids (NWO), Æther (EU) and Apple-CORE (EU). Through the Ph.D. projects, the research in this thesis has directly produced or enabled a total of around 25 publications at the time of this writing. The peer approval of both this research and the subsequently derived and enabled research, together with the apparent appreciation for and external interest in this architecture as shown by these projects further illustrate the successfulness of the developed architecture and this research in particular.

8.3 Future Work

The architecture as described in this thesis is only one aspect of a large system and there was simply not enough time to research and optimize every single aspect of the architecture. As a result, future opportunities for research flow from this research.

8.3.1 Memory

As described in chapter 4, the Microgrid has been designed with a memory network running at 1 GHz that can transmit a 64-byte cache-line from node to node on every cycle. Thus, the maximum theoretical bandwidth of the memory network is 512 Gbps. With DDR3-1600’s maximum theoretical bandwidth of 102.4 Gbps, it takes several DDR3-1600 memory controllers evenly spaced on the top-level ring in the memory network to utilize the maximum memory network bandwidth. In such a case, the entire address space is interleaved in 64-byte blocks across these channels. However, because the memory system was not the focal point of this research, there are a few problems with the current approach.

For instance, the maximum theoretical bandwidth of 102.4 Gbps of a DDR3-1600 channel is based on the situation where the memory pipeline is always full with reads. However, consecutive reads can only be pipelined if they read from the same memory row. If a read has to access another row, the current read first has to complete, the active row has to be precharged, the next row has to be opened, and then the read can be issued; this so-called “read-to-read delay” is typically around 8 times the latency of pipelined reads. Given the concurrent nature of the microgrid and the fact that DDR rows are typically a few kB in size, it is very unlikely to get consecutive reads to the same row when multiple cores are actively working on separate data. In this case the effective bandwidth drops by a factor of 8 to 12.4 Gbps. One possible, though uninvestigated solution to this problem might be a reorder buffer at the memory controller to give requests that access the precharged line priority over earlier requests that access another line.
Another issue is evictions in the memory network. The maximum theoretical bandwidth of the memory network assumes that every message in the network is a read request. However, in the default configuration several cores share a single L2 cache. Depending on the application (e.g. stream processing), there is practically no reuse of the data after it’s entered the L2 cache and the data will soon be evicted to make room for the next cache line. In this case, most reads requests entering the memory system will actually take two cycles because they generate two messages on the network—an eviction message and a read request. This, effectively, halves the maximum achievable bandwidth for reading external data for the memory network.

Additional research should expand on the already available research [97, 98, 120, 79] and investigate alternative implementations. Although the architecture is latency-tolerant, there is no way to get around limited memory bandwidth; the throughput of a memory-heavy application will always be limited by the bandwidth of the memory system.

8.3.2 Operating Systems & Security

There is currently no notion of security or an operating system in the architecture. All code can read and write all data in memory. Page-level security results in large overheads for associative buffers for this access control information and complicates the design of the memory system, especially in a distributed memory. Operating systems issues such as application partitioning, virtual memory and APIs have been side-stepped due to lack of time. These issues must be solved in any general-purpose system. Fortunately, this research has already been started [78].

8.3.3 Exceptions

Just like security, exceptions are also not supported in the current architecture. Not to be confused with external interrupts, which have been covered in section 5.5, exceptions are a signal raised by an instruction executed in the pipeline that something is wrong. Perhaps an input was invalid (e.g., division by zero). Perhaps the program has insufficient rights to perform the action (which ties in with security, above). Either way, something has happened that does not fall under the normal execution of the program and something must now be done. Traditionally, this means flushing the pipeline, saving the state of the program and calling some system-defined exception handler, which has access to the stored state and can, where possible, resume execution after fixing the error. Additionally, some instruction sets also define the notion of precise vs. imprecise exceptions. In the former case, the program must be stopped atomically in a clearly defined place. The design of the pipeline, especially in an out-of-order architecture, complicates this exception handling. In parallel systems these issues are complicated further; does only the single thread get flushed? Does the entire core halt? What are the hardware costs to these approaches? Due to time constraints, these issues have not been included in this thesis, but research on these issues has already started [127, 128].
8.3.4 Preemption

Although the number of cores on a chip keeps growing, there must be a way to pre-empt one or more tasks running on a set of cores in order to free those resources up for another task. The reasons for this can be to ensure responsiveness of all tasks in the case where the number of tasks is greater than the number of execution resources in the entire system, or simply because a task with a higher priority needed to be run.

Preemption requires the state of the current task to be saved, usually to memory or disk, and the resources freed up so it can be reused. In addition to preemption, this mechanism would also allow tasks to be moved across the system. In the case of a microgrid, supporting preemption would not only allow a task to be moved to a set of different cores, but it could also be allocated more or fewer cores.

Although hardware task preemption in the form of termination is supported (see section 4.6), this does not capture state. A possible solution to this problem is to combine termination with software checkpointing [129]. Either way, this scenario has to be investigated further.

8.3.5 Fault Tolerance

A fault tolerant architecture is one where part of the hardware might end up damaged, disabled or otherwise unusable. To avoid such a case rendering the entire processor becoming unusable, the hardware can detect this situation and work around it. Although not a focus of this research, its large number of fungible cores make the microgrid architecture a promising candidate for supporting fault-tolerance. Further research to marry the concept of fault-tolerance to the microgrid is required and has, in fact, already begun [124, 125, 80].
Appendix A

Family Table Contents

This appendix describes the contents of the family table in Microgrid cores. For each field it gives a summary of its purpose, its size in bits and the number of read, write and read/write ports. Bit sizes are specific for a core targeting the Alpha ISA.

<table>
<thead>
<tr>
<th>Name</th>
<th>Purpose</th>
<th>Bits</th>
<th>#Ports</th>
</tr>
</thead>
<tbody>
<tr>
<td>capability</td>
<td>Capability of the family</td>
<td>$S$</td>
<td>2 1 0</td>
</tr>
<tr>
<td>pc</td>
<td>Program counter for new threads</td>
<td>62</td>
<td>2 2 0</td>
</tr>
<tr>
<td>start</td>
<td>Start/Current thread index</td>
<td>64</td>
<td>2 3 1</td>
</tr>
<tr>
<td>step</td>
<td>Thread index step</td>
<td>64</td>
<td>3 3 0</td>
</tr>
<tr>
<td>limit</td>
<td>End thread index</td>
<td>64</td>
<td>0 3 3</td>
</tr>
<tr>
<td>place_size</td>
<td>#cores that this family wanted to run on</td>
<td>$\lceil \log_2 C \rceil$</td>
<td>4 1 0</td>
</tr>
<tr>
<td>num_cores</td>
<td>#cores that this family is running on</td>
<td>$\lceil \log_2 C \rceil$</td>
<td>1 2 1</td>
</tr>
<tr>
<td>block_size</td>
<td>Maximum allocated #threads on this core</td>
<td>$\lceil \log_2 T \rceil$</td>
<td>1 2 2</td>
</tr>
<tr>
<td>alloc_count</td>
<td>Number of family’s threads allocated on this core</td>
<td>$\lceil \log_2 T \rceil$</td>
<td>2 1 3</td>
</tr>
<tr>
<td>alloc_last</td>
<td>Index of the last allocated thread in the family</td>
<td>$\lceil \log_2 T \rceil$</td>
<td>0 1 1</td>
</tr>
<tr>
<td>alloc_done</td>
<td>Is the family done allocating threads?</td>
<td>1</td>
<td>2 2 2</td>
</tr>
<tr>
<td>has_link</td>
<td>Does link contain a valid value?</td>
<td>1</td>
<td>4 1 3</td>
</tr>
<tr>
<td>link</td>
<td>Index of the family’s FT entry on the next core</td>
<td>$\lceil \log_2 F \rceil$</td>
<td>4 1 3</td>
</tr>
<tr>
<td>prev_cleaned_up</td>
<td>Has the last allocated thread been cleaned up?</td>
<td>1</td>
<td>0 1 1</td>
</tr>
<tr>
<td>sync_prev</td>
<td>Has the family on the previous core synchronized?</td>
<td>1</td>
<td>4 1 1</td>
</tr>
<tr>
<td>sync_done</td>
<td>Has the family completed for synchronization?</td>
<td>1</td>
<td>1 3 1</td>
</tr>
<tr>
<td>sync_pid</td>
<td>Address of the the synchronizing thread’s core</td>
<td>$\lceil \log_2 C \rceil$</td>
<td>2 2 1</td>
</tr>
<tr>
<td>sync_reg</td>
<td>Offset of synchronizing register on sync_pid.</td>
<td>$\lceil \log_2 RI \rceil$</td>
<td>2 2 1</td>
</tr>
<tr>
<td>detached</td>
<td>Has the parent thread detached from the family?</td>
<td>1</td>
<td>3 1 2</td>
</tr>
<tr>
<td>pending_reads</td>
<td>Number of pending reads for this family’s threads</td>
<td>$\lceil \log_2 N \rceil$</td>
<td>4 1 2</td>
</tr>
<tr>
<td>int_count_globals</td>
<td>#Global registers in each thread’s integer context</td>
<td>5</td>
<td>3 1 0</td>
</tr>
<tr>
<td>int_count_shared</td>
<td>#Shared registers in each thread’s integer context</td>
<td>5</td>
<td>3 1 0</td>
</tr>
<tr>
<td>int_count_locals</td>
<td>#Local registers in each thread’s integer context</td>
<td>5</td>
<td>3 1 0</td>
</tr>
<tr>
<td>int_base</td>
<td>Register file offset of this family’s integer context</td>
<td>$\lceil \log_2 RI \rceil$</td>
<td>4 1 1</td>
</tr>
<tr>
<td>int_shareds</td>
<td>Register file offset of alloc_last’s integer shares</td>
<td>$\lceil \log_2 RI \rceil$</td>
<td>1 2 1</td>
</tr>
<tr>
<td>flt_count_globals</td>
<td>#Global registers in each thread’s FP context</td>
<td>5</td>
<td>3 1 0</td>
</tr>
<tr>
<td>flt_count_shared</td>
<td>#Shared registers in each thread’s FP context</td>
<td>5</td>
<td>3 1 0</td>
</tr>
<tr>
<td>flt_count_locals</td>
<td>#Local registers in each thread’s FP context</td>
<td>5</td>
<td>3 1 0</td>
</tr>
<tr>
<td>flt_base</td>
<td>Register file offset of this family’s FP context</td>
<td>$\lceil \log_2 RF \rceil$</td>
<td>4 1 1</td>
</tr>
<tr>
<td>flt_shareds</td>
<td>Register file offset of alloc_last’s FP shareds</td>
<td>$\lceil \log_2 RF \rceil$</td>
<td>1 2 1</td>
</tr>
</tbody>
</table>

Some bit sizes are dependent on the architectural configuration and are specified via an abstract constant. Their meaning is listed below:

$C$: Number of cores in the Microgrid.
$T$: Number of thread table entries in a core.
$F$: Number of family table entries in a core.
$RI, RF$: Number of {integer, FP} registers in a core.
$S$: Size of a family’s capability.
$N$: Worst-case number of reads pending by a single thread ($N \leq RI + RF$).
Appendix B

Thread Table Contents

This appendix describes the contents of the thread table in Microgrid cores. For each field it gives a summary of its purpose, its size in bits and the number of read, write and read/write ports that have been simulated. Note that non-boolean explicit bit sizes are specific for a Microgrid core targeting the Alpha instruction set.

<table>
<thead>
<tr>
<th>Name</th>
<th>Purpose</th>
<th>Bits</th>
<th>#Ports</th>
</tr>
</thead>
<tbody>
<tr>
<td>pc</td>
<td>Current program counter</td>
<td>62</td>
<td>2 2 0</td>
</tr>
<tr>
<td>int_locals</td>
<td>Integer register file offset of thread’s locals</td>
<td>$\lceil \log_2 RI \rceil$</td>
<td>1 1 0 1</td>
</tr>
<tr>
<td>int_dependents</td>
<td>Integer register file offset of thread’s dependents</td>
<td>$\lceil \log_2 RI \rceil$</td>
<td>1 1 0 0</td>
</tr>
<tr>
<td>int_shared</td>
<td>Integer register file offset of thread’s shareds</td>
<td>$\lceil \log_2 RI \rceil$</td>
<td>1 1 0 0</td>
</tr>
<tr>
<td>flt_locals</td>
<td>FP register file offset of thread’s locals</td>
<td>$\lceil \log_2 RF \rceil$</td>
<td>1 0 1 0</td>
</tr>
<tr>
<td>flt_dependents</td>
<td>FP register file offset of thread’s dependents</td>
<td>$\lceil \log_2 RF \rceil$</td>
<td>0 1 0 1</td>
</tr>
<tr>
<td>flt_shared</td>
<td>FP register file offset of thread’s shareds</td>
<td>$\lceil \log_2 RF \rceil$</td>
<td>1 1 0 0</td>
</tr>
<tr>
<td>killed</td>
<td>Has the thread terminated?</td>
<td>1</td>
<td>1 1 1</td>
</tr>
<tr>
<td>prev_cleaned_up</td>
<td>Has the thread’s predecessor been cleaned up?</td>
<td>1</td>
<td>1 1 1</td>
</tr>
<tr>
<td>pending_writes</td>
<td>Number of unacknowledged writes made by this thread</td>
<td>$\lceil \log_2 W \rceil$</td>
<td>0 1 3</td>
</tr>
<tr>
<td>sequence_next</td>
<td>Thread table index of this thread’s successor</td>
<td>$\lceil \log_2 T \rceil$</td>
<td>0 1 2</td>
</tr>
<tr>
<td>cid</td>
<td>I-Cache line containing the thread’s instructions</td>
<td>$\lceil \log_2 L \rceil$</td>
<td>2 1 0</td>
</tr>
<tr>
<td>family</td>
<td>Family table index of this thread’s family</td>
<td>$\lceil \log_2 F \rceil$</td>
<td>1 0 1</td>
</tr>
<tr>
<td>next</td>
<td>Next thread in the linked list</td>
<td>$\lceil \log_2 T \rceil$</td>
<td>1 1 1</td>
</tr>
</tbody>
</table>

Some bit sizes are dependent on the architectural configuration and are specified via an abstract constant. Their meaning is listed below:

$L$: Number of lines in the core’s I-Cache.

$T$: Number of thread table entries in the core.

$F$: Number of family table entries in the core.

$RI,RF$: Number of \{integer, FP\} registers in a core.

$W$: Worst-case number of writes pending by a single thread.
Appendix C

Distribute Dependent Families

This appendix aims to analyze the difference between distributing dependent families (i.e., families with one or more shareds) across more than one core, or running the entire family on a single core.

We assume a thread is defined by these consecutive regions of code: \texttt{Start}; \texttt{Rs}; \texttt{Work}; \texttt{Ws}; \texttt{End}. The \texttt{Rs} and \texttt{Ws} operations read and write a shared, respectively. The time taken for these operations are insignificant next to the other three section of the thread. Therefore, a thread can be described by the three timing values: \(T_s\), \(T_w\) and \(T_e\) and as a convenience, their sum \(T_{sw}e\).

Assuming the time to transfer dependencies and reuse and allocate a thread is insignificant, we can model the execution time of a family of threads (\(\Omega\)) solely by \(T_s\), \(T_w\), \(T_e\), the number of threads in the family (\(N\)), the block size (\(B\)) and the number of cores across which it is distributed (\(P\)).

We will examine two methods:

1. the threads are distributed round-robin by block size \(B\) over \(P\) cores.
2. the threads are run on a single core with a block size \(B\).

For each method, we will derive an equation for the total execution time of a family and in the end compare the two methods to see which conditions need to apply for one method to be better (i.e., have a smaller execution time) than another. Note that in the sections below, the term generation refers to the set of threads that have forward-only communication (i.e., the last thread in a generation communicates with the first thread in the next generation via a “wrap-around”).

Finally, note that the following conditions always hold: \(P > 1\), \(B \geq 1\), \(T_s \geq 0\), \(T_w \geq 0\) and \(T_e \geq 0\).

C.1 Method 1: Distribute

This section analyzes the case where a dependent family is distributed across \(P\) cores. As illustrated in figure [C.1] in this method \(P \cdot B\) threads start at the same time. All threads run in parallel for \(T_s\) amount of time. Then, the first thread can read its shared immediately. We define the time at which the first thread in the first
Figure C.1: Example execution for method 1. Shown are 12 threads running on 2 cores with a block size of 3 on each, creating 2 generations of threads. The gaps after $R_s$ in most threads is that thread waiting for its predecessor to write its shared.
Appendix C. Distribute Dependent Families

generation can start its work as $S_1$:

$$S_1 = T_s$$

The thread then continues on for $T_w$ amount of time and writes its shared. At this point, the second thread can continue with $T_w$ amount of work after which it writes its shared, and so on. Since $P \cdot B$ threads exist in the generation, we can define the time at which the last thread in the first generation writes its shared as $W_1$:

$$W_1 = S_1 + P \cdot B \cdot T_w$$

The first thread in the second generation runs when the first thread in the first generation has completed, after $S_1 + T_w + T_e$ amount of time. The new thread can read its shared at time $S_1 + T_w + T_e + T_s$, although it may not be available yet. Thus, the first thread in the second generation starts work at time $S_2$:

$$S_2 = \max(S_1 + T_w + T_e + T_s, W_1)$$
$$= \max(S_1 + T_w + T_e + T_s, S_1 + P \cdot B \cdot T_w)$$
$$= S_1 + \max(T_w + T_e + T_s, P \cdot B \cdot T_w)$$
$$= S_1 + \max(T_{swe}, P \cdot B \cdot T_w)$$

Similarly, the first thread in the third generation starts at $S_3 = S_2 + \max(T_{swe}, P \cdot B \cdot T_w)$. We can generalize this into the time that the first thread in generation $n$ starts work:

$$S_n = T_s + (n - 1) \max(T_{swe}, P \cdot B \cdot T_w)$$

We define the time that the last thread in the last generation completes as $C_n = W_n + T_e$. With $\frac{N}{P \cdot B}$ generations in the family, the total time for the family is:

$$\Omega_1 = C_{\frac{N}{P \cdot B}}$$
$$= W_{\frac{N}{P \cdot B}} + T_e$$
$$= S_{\frac{N}{P \cdot B}} + P \cdot B \cdot T_w + T_e$$
$$= T_s + (\frac{N}{P \cdot B} - 1) \max(T_{swe}, P \cdot B \cdot T_w) + P \cdot B \cdot T_w + T_e$$

C.2 Method 2: Do not distribute

This section analyzes the case where a dependent family is run on a single core ($P = 1$). As shown in figure C.2, in this method $B$ threads start at the same time. All threads run in parallel for $T_s$ amount of time. Then, the first thread can read its shared immediately. We define the time at which the first thread in the first generation can start its work as $S_1$:

$$S_1 = T_s$$
Figure C.2: Example execution for method 2. Shown are 12 threads running on 1 core with a block size of 3, creating 4 generations of threads. The gaps after $R_s$ in most threads is that thread waiting for its predecessor to write its shared.
Appendix C. Distribute Dependent Families

The thread then continues on for \( T_w \) amount of time and writes its shared. At this point, the second thread can continue with \( T_w \) amount of work after which it writes its shared, and so on. Since \( B \) threads exist in the generation, we can define the time at which the last thread in the first generation writes its shared as \( W_1 \):

\[
W_1 = S_1 + B \cdot T_w
\]

The first thread in the second generation runs when the first thread in the first generation has completed, after \( S_1 + T_w + T_e \) amount of time. The new thread can read its shared at time \( S_1 + T_w + T_e + T_s \), although it may not be available yet. Thus, the first thread in the second generation starts work at time \( S_2 \):

\[
S_2 = \max(S_1 + T_w + T_e + T_s, W_1)
= \max(S_1 + T_w + T_e + T_s, S_1 + B \cdot T_w)
= S_1 + \max(T_w + T_e + T_s, B \cdot T_w)
= S_1 + \max(T_{swe}, B \cdot T_w)
\]

Similarly, the first thread in the third generation starts at \( S_3 = S_2 + \max(T_{swe}, B \cdot T_w) \).

We can generalize this into the time that the first thread in generation \( n \) starts work:

\[
S_n = T_s + (n - 1) \max(T_{swe}, B \cdot T_w)
\]

We define the time that the last thread in generation \( n \) completes as \( C_n = W_n + T_e \). With \( \frac{N}{B} \) generations in the family, the total time for the family is:

\[
\Omega_2 = C_n
= W_n + T_e
= S_n + B \cdot T_w + T_e
= T_s + (\frac{N}{B} - 1) \max(T_{swe}, B \cdot T_w) + B \cdot T_w + T_e
\]

C.3 Comparison

Since the second method is easier to support in hardware, we want to see under what conditions \( \Omega_2 \leq \Omega_1 \).

To establish which situations allow for this, we recognize that there are several scenarios to consider, due to the function \( \max \) being present in the time equations. There are two \( \max \) functions to consider: \( \max(T_{swe}, P \cdot B \cdot T_w) \) from method 1 and \( \max(T_{swe}, B \cdot T_w) \) from method 2. We can construct three scenarios to fully cover all outcomes of the \( \max \) functions.

1. \( T_{swe} \leq B \cdot T_w \)
2. \( T_{swe} \geq P \cdot B \cdot T_w \)
3. \( B \cdot T_w < T_{swe} < P \cdot B \cdot T_w \)
Appendix C. Distribute Dependent Families

Since $\Omega_1$ and $\Omega_2$ are both of the form $\ldots + T_s + T_e$ (i.e., there’s always the minimum latency of the Start and End section), the two terms at the end have already been removed before the comparisons below. Finally, we can assume that $N > B$, otherwise the family will never be distributed even in method 1.

C.3.1 Scenario 1

Assuming that $T_{swe} \leq B \cdot T_w$, then

$$\Omega_2 \leq \Omega_1$$

$$\left(\frac{N}{B} - 1\right) \max(T_{swe}, B \cdot T_w) + B \cdot T_w \leq \left(\frac{N}{P \cdot B} - 1\right) \max(T_{swe}, P \cdot B \cdot T_w) + P \cdot B \cdot T_w$$

$$\left(\frac{N}{B} - 1\right)(B \cdot T_w) + B \cdot T_w \leq \left(\frac{N}{P \cdot B} - 1\right)(P \cdot B \cdot T_w) + P \cdot B \cdot T_w$$

$$\frac{N}{B} B \cdot T_w \leq \frac{N}{P \cdot B} (P \cdot B \cdot T_w)$$

$$N \cdot T_w \leq N \cdot T_w$$

$$0 \leq 0$$

This is a tautology and thus, in this scenario, it is always preferable to use the second method. Note that the reverse ($\Omega_1 \leq \Omega_2$) is also always true, so technically it doesn’t matter which method is used.

C.3.2 Scenario 2

Assuming that $T_{swe} \geq P \cdot B \cdot T_w$ then

$$\Omega_2 \leq \Omega_1$$

$$\left(\frac{N}{B} - 1\right) \max(T_{swe}, B \cdot T_w) + B \cdot T_w \leq \left(\frac{N}{P \cdot B} - 1\right) \max(T_{swe}, P \cdot B \cdot T_w) + P \cdot B \cdot T_w$$

$$\left(\frac{N}{B} - 1\right)(T_{swe}) + B \cdot T_w \leq \left(\frac{N}{P \cdot B} - 1\right)(T_{swe}) + P \cdot B \cdot T_w$$

$$\frac{N}{B} T_{swe} + B \cdot T_w \leq \frac{N}{P \cdot B} T_{swe} + P \cdot B \cdot T_w$$

$$\frac{N}{B} (T_{swe}) - \frac{N}{P \cdot B} (T_{swe}) \leq P \cdot B \cdot T_w - B \cdot T_w$$

$$\frac{P \cdot N}{P \cdot B} (T_{swe}) \leq (P - 1) \cdot B \cdot T_w$$

$$(P - 1) \frac{N}{P \cdot B} (T_{swe}) \leq (P - 1) \cdot B \cdot T_w$$

$$\frac{N}{P \cdot B} (T_{swe}) \leq B \cdot T_w$$

$$\frac{N}{B} \leq \frac{P \cdot B \cdot T_w}{T_w + T_e + T_s}$$

However, given the initial assumption of $T_{swe} \geq P \cdot B \cdot T_w$, or $\frac{P \cdot B \cdot T_w}{T_{swe}} \leq 1$, that means that at least $\frac{N}{B} \leq 1$, which means that $N \leq B$. However, our initial assumption for distribution to make sense is that $N > B$. Therefore, it is never better to use method 2 in this scenario.
C.3.3 Scenario 3

Assuming that $B \cdot T_w < T_{swe} < P \cdot B \cdot T_w$ then

$$\Omega_2 \leq \Omega_1$$

$$(\frac{N}{B} - 1) \max(T_{swe}, B \cdot T_w) + B \cdot T_w \leq (\frac{N}{P \cdot B} - 1) \max(T_{swe}, P \cdot B \cdot T_w) + P \cdot B \cdot T_w$$

$$(\frac{N}{B} - 1)(T_{swe}) + B \cdot T_w \leq (\frac{N}{P \cdot B} - 1)P \cdot B \cdot T_w + P \cdot B \cdot T_w$$

$$(\frac{N}{B} - 1)(T_{swe}) + B \cdot T_w \leq N \cdot T_w$$

$$(\frac{N}{B} - 1)(T_{swe}) \leq (N - B) \cdot T_w$$

$$\frac{N - B}{B}(T_{swe}) \leq (N - B) \cdot T_w$$

$$T_{swe} \leq B \cdot T_w$$

However, given the initial assumption of $T_{swe} > B \cdot T_w$, this condition is always false. Thus, in this scenario, the first method is always preferable.

C.4 Analysis

Scenario 1 shows that if the length of a thread ($T_{swe}$) is less than the time it takes to execute a single block’s worth of work on a single core ($B \cdot T_w$), both methods will result in the same family execution time. This result can be understood in that, in method 2, the first thread of the next generation will be ready to receive its shared from the last thread in the previous generation. Thus, there are no unnecessary delays. And since method 1 has more tolerance towards the length of the thread (since it executes a factor $P$ more threads before the shared comes to the first thread in the next generation), it also suffers no unnecessary delays. Thus, the total execution time of the family in both cases is the time for the serialized work ($N \cdot T_w$), plus the start and end time for the first and last thread, respectively.

Scenario 3 shows that if the length of a thread is more than the time it takes to execute a block’s worth of work on all cores, method 1 is absolutely advantageous. Using the same reasoning as above, this also comes as no surprise. In this scenario, the second method will cause the first thread in the next generation to wait for the first thread in the last generation to finish before it can accept its shared. Thus, the serialized path ($N \cdot T_w$) is delayed. With method 1, however, the serialized path is not delayed, as the first thread of the next generation is ready to accept the shared. This difference comes from the fact that in the first method, the longer duration of a thread is amortized by having more threads to run before that thread’s slot is reused.

Scenario 2 shows that if the length of a thread is more than the time it takes for a block’s worth of work on all cores, the choice of distribution depends on the number of threads, number of cores and the block size. Note that in this case, even method 1 suffers a delay from the sequential path, just like method 2 did in the previous scenario.
C.5 Conclusion

We can rewrite the three scenario conditions, combined with the results, as follows:

1. $\frac{T_{swe}}{T_{w}} \leq B$: Both methods are equally good.

2. $\frac{T_{swe}}{T_{w}} \geq P \cdot B$: Method 1 is better.

3. $B < \frac{T_{swe}}{T_{w}} < P \cdot B$: Method 1 is better.

Or, put another way, if $\frac{T_s + T_w}{T_w} > B - 1$ then method 1 is better, otherwise it doesn’t matter. So, strictly speaking it seems that distributing the threads is never worse than not distributing. However, having hardware that is able to distribute threads with proper inter-core memory consistency is more expensive, and method 1 is only better when $\frac{T_s + T_w}{T_w} > B - 1$. This means that, for instance, with a block size of 4, distributing threads over multiple cores is only beneficial if the combined prefix and postfix part of the threads last 3x longer than the sequentialized part. The larger the block size, the worse the effect.

Thus, **it is only advantageous to distribute a family with shareds across multiple cores if the postfix and prefix sections of the thread last significantly long compared to the work section.** However, since the postfix and prefix sections are, by definition, independent of each other, such a family can always be split up into three families: one independent prefix family, one dependent work family (with negligible prefix and postfix sections) and one independent postfix family.

As a result, it is practically unnecessary to support distributing dependent families across multiple cores in hardware.
Appendix D

Instructions set additions

The hardware architecture described in chapter 4 is based on family creation, synchronization and management with dedicated instructions that have been added to the instruction set of an existing RISC-like ISA. This appendix describes, for completeness, the added instructions, which operands they take and what function they perform. The instructions listed below comprise the main functionality of the microgrid extensions to a RISC core.

D.1 Allocate

The allocate instruction allocates a single family context across the cores on a place. A family context consists of a single family table entry, one thread table entry and enough registers to store at least one thread’s context assuming worst-case register usage. This way, after the allocate instruction has completed successfully, the program can be sure that any family it places there will not suffer from resource deadlock due to the fact that the intra-family synchronization mechanisms in the execution model permit the entire family to be run sequentially, one thread at a time.

The instruction has as input the identifier (see section 5.2) of the place where the context should be allocated and has one output operand that will receive a reference to the allocated context, or 0 if the allocation fails. This instruction is a long-latency instruction that clears its output register upon issuing. Once the allocate process has completed (successfully or not), the context reference, or 0, is written back into the register.

Optionally, the allocate instruction can have an exclusive flag or an suspend flag. The exclusive flag will make the allocation use the core’s exclusive context (see section 4.5.5). The suspend flag will cause the allocation to queue on the destination core until a context is available. Omitting this flag causes the allocate to fail if no contexts are free on the destination core. A non-suspending create can be used by software to avoid resource deadlock (see section 4.7).
D.2 Set Start/Limit/Step

The \texttt{setstart}, \texttt{setlimit} and \texttt{setstep} instructions setup the family index sequence (see section 3.1.1) of a successfully allocated context. Each of the three instructions has two inputs: one input operand containing the reference to the family context whose index sequence to set up, and one input operand containing the value to write to the start, limit, or step field, respectively. These instructions can only be issued between the “allocate” and “create” instructions mentioned above and below, respectively. They are, however, idempotent, and can be issued multiple times to overwrite the previously set values. To allow for efficient creation of functional concurrency, the defaults of the start, limit and step are 0, 1 and 1, respectively. This creates a family of one thread allowing a function to be created concurrently with minimal instructions.

D.3 Set Block

The \texttt{setblock} instruction sets the block size (see section 3.1.2) of a successfully allocated context. This instruction has one input operand containing the reference to the family context whose block size to set, and one input operand containing the new value of the block size. If this instruction is not executed, the family retains its default block size, 0, which is replaced with the thread table size upon family creation. This instruction can only be issued between the “allocate” and “create” instructions mentioned above and below, respectively. It is, however, idempotent, and can be issued multiple times to overwrite the previously set value.

D.4 Create

The \texttt{create} instruction sets the program counter of the family context and signals its creation at the same time. This instruction has one input operand containing the reference to the family context whose create to signal, one input operand containing the program counter for the threads in the family and one output operand that will receive the family identifier of the family (i.e., a copy of in the input operand) when the family has been created and its arguments can be sent. This instruction both writes the program counter to the allocated context(s) and marks the family ready for thread creation.

This instruction is a long-latency instruction that clears its output register upon issuing. Once the create process has completed, meaning the registers have been allocated and the threads can be created and the arguments written, the family identifier is written back into the register.

D.5 Put Global/Shared

The \texttt{putg} and \texttt{puts} instructions write a register value into the globals or input of the dependency chain of a created family, respectively. Both instructions have one
input operand containing the reference to the family whose global or first dependent
to write, one literal identifying which global or shared to write and one input operand
containing the value to write. These instructions can only be issued after the family
has been created with the “create” instruction described above.

The globals and first thread’s dependents start off as empty when the family is
created, so when a thread attempts to read them, it will suspend on them. Thus,
there is no time constraint for writing these arguments after the family has been
created. However, after writing a global or shared, it should not be written again
(with a different value) as this might produce non-deterministic behavior: the parent
thread cannot know which value the child thread(s) use as this depends entirely on
timing.

D.6 Get Shared

The gets instruction reads the output of the dependency chain of a terminated family.
The instruction has three operands: one input operand containing the family identifier
of the family whose last dependent to read, one input operand identifying which shared
to read and one output register that will receive the read value. This instruction can
only be issued after the family has been synchronized upon with the “sync” instruction
described above.

The gets instruction is a long-latency instruction that clears its output register
upon issuing, since the act of retrieving the output of the dependency chain will most
likely several cycles, depending on the distance of the family from the issuing thread.
Once the value is retrieved, the output register is written.

D.7 Synchronize

The sync instruction signals a created family that the issuing thread wants to syn-
chronize (see section 3.1.3) on its termination. The instruction has an input operand,
containing the family identifier of the family it wishes to synchronize upon and an
output operand, which will be written when the specified family has synchronized.
This instruction sends a message to the target family, notifying it of its desire to
synchronize on it, along with a reference to the output register. If, upon arrival of the
message, the family has already completed, a message is sent back that will write an
undetermined value to the output register. Otherwise, the reference to the register is
stored in the family context, so that the message can be sent when the family com-
pletes. This instruction can only be issued after the family has been created with the
“create” instruction described above and should be issued only once for the specified
family.

Note that issuing this instruction will not suspend the thread until the specified
family has completed. It only notifies the specified family and sets up a reference to
the output register. The thread should read this instruction’s output register after
issuing this instruction to actually suspend until the family has terminated.

The sync instruction is a long-latency instruction that clears its output register
upon issuing. Once the specified family has completed, meaning all threads have
terminated and all memory writes are consistent, the output register is written.

D.8 Detach

The *detach* instruction signals a created family that the issuing thread is no longer interested in it. This means that once the specified family has terminated, it can be cleaned up as no further requests will be made to it (see section 4.5.3). This instruction has only a single input operand containing the family identifier of the family it wishes to detach from. *detach* is typically the last instruction for a family to be executed, usually after the *sync* and optionally *gets* instructions described above. Failing to issue this instruction will cause a resource leak that could lead to deadlock or severely reduced performance once the hardware’s physical resources have been exhausted.

Note that this instruction can be issued immediately after the *create* instruction described above. In such an event, the thread can no longer synchronize on the family’s completion or read its output dependents.

D.9 Break

The *break* instruction halts creation of threads in the current family (see section 4.6). Existing threads can still run to completion. This instruction has no operands and is used to cooperatively terminate the current family. It can be used by programs to implement dynamic termination conditions on loops implemented as families.

D.10 Kill

The *kill* instruction atomically terminates every family and thread on a place (see section 4.6). This instruction has a single operand containing the identifier of the place that should be killed. The *kill* instruction implements pre-emptive family termination on place granularity.
Publications

This chapter lists all publications that are directly representative of the research described in this thesis, as well as publications that exploit the research and would not have been possible without it.

Representative publications

Listed below are the publications that directly represent the work and research described in this thesis, as well as a summary of the influence of the research described in this thesis. Please note that the authors are ordered alphabetically where noted, or else editor first followed by main research contributors. For clarity, the individual contributions of the authors have been listed for each publication.

Individual author contributions: M. Lankamp: 45% (architecture design, simulation tools design and implementation, benchmark authoring, evaluation), R. Poss: 30% (programming tools, operating system components), Q. Yang: 10% (memory protocol optimizations) J. Fu: 5% (support to evaluation) M.W. van Tol: 5% (operating system directions) I. Uddin: 5% (support to simulation). Directly reflects chapters 3, 4, 5 and 7 of this thesis, and parts of chapter 6.

Individual author contributions: M. Lankamp: 70% (section II, III, IV, VI and VII), R. Poss: 10% (section V and VI.D), Q. Yang: 10% (1 memory model among the 6 implemented), J. Fu: 5% (support to evaluation), I. Uddin: 5% (support to simulation) Directly reflects chapter 6 of this thesis, and parts of chapter 7.

Individual author contributions same as [P.2].
Individual author contributions same as [P.1]

Individual author contributions: M. Lankamp: 30% (system strategy, realization in low-level software simulation, evaluation), R. Poss: 30% (evaluation requirements, system strategy, programming tools, I/O components in simulation), J. Sykora and L. Kafka: 25% (realization in FPGA), M. Irfan Uddin: 15% (realization in high-level software simulation). Directly reflects chapter 6 of this thesis, and parts of chapter 7.

(Author list in alphabetical order) Individual author contributions: M. Lankamp: 60% (section 3 and 4, architecture design, simulation tools, benchmark implementation and result analysis), L. Zhang: 20% (memory protocol to support evaluation in section 4), C. Jesshope: 10% (section 2.1), M. Hicks: 5% (code implementation for section 4.1, plot generation), R. Poss: 5% (section 2.2, memory model). Directly reflects chapters 3, 4, 5 and 7 of this thesis, and parts of chapter 6.

(Author list in alphabetical order) Individual author contributions: M. Lankamp 50% (architecture definition, simulation design and implementation, benchmark implementation and result analysis), L. Zhang: 30% (COMA memory protocol), C. Jesshope: 20% (section 2, abstract model). Directly reflects chapters 3, 4, 5 and 7 of this thesis, and parts of chapter 6.

Individual author contributions same as [P.7]

Individual author contributions: M.W. van Tol: 75% (main investigation), M. Lankamp: 15% (design directions for the architecture model being simulated by the POSIX simulation presented in the article), S. Polstra: 10% (support to evaluation). Reflects parts of chapters 3 and 4 of this thesis.


Individual author contributions: M. Lankamp: 35% (processor design, low-level simulation, evaluation), K. Bousias: 35% (model design, functional simulation), L. Guang: 30% (memory architecture design and prototyping). Directly reflects chapters 3, 4 and 7 of this thesis, and parts of chapter 6.


Individual author contributions same as P.6.


Individual author contributions: C. Joslin: 50% (main compiler implementation), M. Lankamp: 20% (provided the target architecture definition and an execution platform in the form of simulations, feedback on evaluation), S. Herhut: 20% (abstract compiler transformations), C. Grelck and S.B. Scholz: 10% (design directions, supervision). Reflects parts of chapters 3, 4 and 5 of this thesis, and reuses outcomes of chapter 6.


Individual author contributions: M.W. van Tol: 75% (main investigation), M. Lankamp: 15% (design directions for the architecture model being simulated by the POSIX simulation presented in the article), S. Polstra: 10% (support to evaluation). Reflects parts of chapters 3 and 4 of this thesis.

**Enabled publications**

Listed below is a list of publications that have been enabled by the work and research described in this thesis. They all base their design and evaluation on the architec-


Bibliography


[74] OpenMP ARB. OpenMP. Website, 2013. URL http://openmp.org/


Performance improvements for microprocessors have traditionally been achieved by increasing their clock frequency. However, this technique has reached a point where further scaling is impractical.

This thesis describes and evaluates a novel System-on-Chip architecture that focuses on exploiting all forms of concurrency in programs. It does so by defining generic hardware concurrency management extensions in simple multi-threaded cores. These extensions enable low-overhead bulk-creation and dynamic distribution of threads and expose efficient dataflow-like primitives for both inter-thread and intra-thread communication and synchronization.

Additionally, this thesis describes a new cycle-accurate processor architecture software simulator written in C++, in a way to make it a valuable research and education tool by allowing for a clean and relatively high-level description of the architecture.