Technical Report

UCAM-CL-TR-846
ISSN 1476-2986

Number 846

UNIVERSITY OF CAMBRIDGE
Computer Laboratory

Exploiting tightly-coupled cores

Daniel Bates

January 2014

15 JJ Thomson Avenue
Cambridge CB3 0FD
United Kingdom
phone +44 1223 763500

http://www.cl.cam.ac.uk/

This technical report is based on a dissertation submitted July 2013 by the author for the degree of Doctor of Philosophy to the University of Cambridge, Robinson College.

Technical reports published by the University of Cambridge Computer Laboratory are freely available via the Internet:

http://www.cl.cam.ac.uk/techreports/

ISSN 1476-2986

ABSTRACT

As we move steadily through the multicore era, and the number of processing cores on each chip continues to rise, parallel computation becomes increasingly important. However, parallelising an application is often difficult because of dependencies between different regions of code which require cores to communicate. Communication is usually slow compared to computation, and so restricts the opportunities for profitable parallelisation. In this work, I explore the opportunities provided when communication between cores has a very low latency and low energy cost. I observe that there are many different ways in which multiple cores can be used to execute a program, allowing more parallelism to be exploited in more situations, and also providing energy savings in some cases. Individual cores can be made very simple and efficient because they do not need to exploit parallelism internally. The communication patterns between cores can be updated frequently to reflect the parallelism available at the time, allowing better utilisation than specialised hardware which is used infrequently.

In this dissertation I introduce Loki: a homogeneous, tiled architecture made up of many simple, tightly-coupled cores. I demonstrate the benefits in both performance and energy consumption which can be achieved with this arrangement and observe that it is also likely to have lower design and validation costs and be easier to optimise. I then determine exactly where the performance bottlenecks of the design are, and where the energy is consumed, and look into some more-advanced optimisations which can make parallelism even more profitable.

ACKNOWLEDGEMENTS

First, I would like to thank my supervisor, Robert Mullins, without whose help, support and insight, this work would not have been possible.

I would also like to thank David Greaves and Steven Hand for their role in guiding me towards an achievable and worthwhile goal over the last few years.

The rest of Team Loki—Alex Bradbury, Andreas Koltes and George Sarbu—have provided countless interesting discussions, whether work-related or not, and I am indebted to them for their work on other parts of the Loki design and infrastructure. Other members of the Computer Architecture Group have also been quick to give advice and feedback when I needed it.

Thanks must go to the Raspberry Pi Foundation – I admire their aim of getting more people interested in Computer Science, and it has been a privilege to be as involved as I have been. I am also thankful for the opportunities they provided to take the occasional break from thesis-writing and meet some very interesting people.

My research was funded with a grant from the Engineering and Physical Sciences Research Council.

CONTENTS

LIST OF FIGURES

LIST OF TABLES

CHAPTER 1

INTRODUCTION

This dissertation describes Loki, a homogeneous, many-core architecture targeted at embedded systems. One of Loki's main features is flexible and efficient communication between cores, allowing them to be grouped together in software to form an optimised *virtual architecture*. This software specialisation using tightly-coupled cores allows both improved performance and reduced energy consumption, whilst also addressing many other challenges faced by today's computer architects.

1.1 Motivation

Multi-core chips dominate the consumer market today. Everything from smartphones, through tablets and desktops, all the way up to servers make use of multi-core processors, and the number of cores continues to rise as architects struggle to make effective use of the extra transistors they gain each process generation. A group from the University of California, Berkeley describe the situation well [12]:

> The shift toward increasing parallelism is not a triumphant stride forward based on breakthroughs in novel software and architectures for parallelism; instead, this plunge into parallelism is actually a retreat from even greater challenges that thwart efficient silicon implementation of traditional uniprocessor architectures.

The problem with the trend towards multiple cores is that even after decades of research into parallelism, many applications do not make effective use of parallel architectures [21], and it seems unlikely that the number of applications executing simultaneously will scale with the number of processor cores. Work has been done on cache coherence and message passing to make using multiple cores easier, but more needs to be done if the full potential of future processors is to be realised.

One of the main challenges is the power wall: the inability to increase power consumption of processors due to limitations of the power supply (such as batteries in mobile devices) or the ability to dissipate heat cheaply. Much of the exponential performance improvements of microprocessors that we have enjoyed to date came at a cost of increased power consumption; this was a reasonable tradeoff at the time, but cannot continue today. Several factors conspire to make the power wall even more intimidating:

- As we continue to shrink features on integrated circuits, the energy consumed by each transistor goes down, but the delay and energy consumed per unit length of wire go up.

The delay rises due to increased resistance of the narrower wires, and so more power-hungry repeaters are required to maintain clock rates [63]. This suggests that large uniprocessors are no longer sensible and a move to decentralised resources is necessary in order to reduce communication distances [6].

- The clock frequency of a microprocessor is intimately linked with power consumption. Each change of state of a transistor or wire costs roughly the same amount of energy, but a higher frequency means that the state can change more often, increasing power consumption. Many techniques employed to increase clock frequencies also bring with them additional energy costs: deeper pipelining, for example, requires additional pipeline registers and more-aggressive circuits. Since this power increase is no longer an option, clock frequencies of desktop and server chips have stalled, along with the associated increase in performance. Other ways of improving performance, such as exploiting instruction-level parallelism, are also near their limits: ever more complex branch prediction and increased speculation are required to find further independent instructions, both of which require energy.

- Dennard scaling—the ability to reduce the supply voltage as transistors become smaller—has essentially stopped. Power consumed when a transistor switches is proportional to the square of the voltage, so the regular decline in voltage each generation held the power wall at bay. Any attempts to reduce the threshold voltage of transistors to allow a lower supply voltage result in increased leakage current. Leakage power has quickly gone from a negligible overhead to a source of energy consumption comparable with the total energy used by active parts of the chip.

- Smaller transistors are less reliable due to fluctuations in the density of dopant atoms in the underlying silicon [22]. The obvious solution to unreliable systems is redundancy, but this involves performing additional computation, which increases energy requirements.

A consequence of the power wall in combination with the ever-increasing number of transistors available due to Moore's Law is that we may no longer be able to use all of the available transistors at the same time. This problem is known as the *utilisation wall* or *dark silicon*, and implies that some form of specialisation will be required to reduce the fraction of the chip in use at any one time [37]. A move towards specialised hardware is also desirable because of the relatively high overheads of general-purpose chips [51].

Indeed, modern chips often bundle a number of different specialised processors, cache, and other logic together, usually with a general purpose processor to fall back on when no accelerator is available. The result is a *system-on-chip* (SoC) capable of general purpose computation, but which can achieve high performance and energy efficiency for many common operations such as graphics and signal processing. This approach has seen success in the mobile market as the reduction in the number of chips reduces circuitboard area and packaging costs, and is also gaining popularity in the desktop landscape.

The move to heterogeneous computing will likely provide only a short reprieve from the challenges faced by computer architects, however, and I believe that the balance will start to swing back towards homogeneity in the near future.

- Design and validation costs are increasing rapidly as we move to smaller feature sizes. Including many different specialised units will only increase these costs, and managing communication between them complicates the design further.

- Providing many different processing units means that many different forms of fault tolerance need to be implemented when we cannot rely on all transistors working correctly.

- Heterogeneous systems are more difficult to program, and as software and algorithms become increasingly complex, this becomes less acceptable.

Computer architects are trapped by all of the constraints and walls: they can't dynamically extract more ILP, they can't use additional power to improve performance, they can't scale the voltage down to reduce power consumption, and they can't add specialised hardware because it will soon become too difficult to design and use. The only escape route seems to be to make a step-change in the design of microprocessors – one which takes into account *all* of the current challenges, rather than making incremental changes to decades-old designs.

Computer architecture is often described as a science of tradeoffs: there are many different metrics for which a design can be optimised, most of which have conflicting requirements. Smartphone consumers demand the most powerful hardware capable of advanced 3D graphics, and a large, bright, high resolution screen, whilst simultaneously complaining if battery life drops. The problem is that these tradeoffs are not static: processors designed today face completely different constraints to those designed only a decade ago. Momentum in industry dictates that processor designs are evolutionary rather than revolutionary in order to maintain backwards-compatibility, but this direction appears to be leading us to a dead-end.

In this dissertation, I attempt to take a step back and look ahead to determine what a processor should look like in the near future, in a world where high performance is needed with low power consumption. Specialisation will be required to achieve efficiency, but sacrificing programmability is unacceptable. Furthermore, we will not be able to guarantee that all of the chip will behave as expected, and design and validation costs are expected to continue rising exponentially.

To this end, I introduce Loki, a many-core homogeneous architecture, where cores are very simple and designed to be dynamically grouped together to execute a program. Flexible and efficient communication between cores is key, and allows for specialisation in software. This design attempts to address the issues imposed by Amdahl's law by allowing multiple cores to be used to accelerate both parallel and sequential regions of code. I demonstrate how tight coupling between cores can be used both to improve performance and to reduce energy consumption.

1.2 Contributions

My thesis is that efficient communication between processor cores provides opportunities for flexible parallelism, which can be profitable in terms of both energy and performance. In order to demonstrate this, I make the following contributions:

- I *introduce the Loki architecture*: a simple, homogeneous, many-core fabric, where communication between cores is fast and energy-efficient (Chapter 3).

- I provide *detailed energy, area and performance models* of the main components of the design, and show where the energy is consumed during execution (Chapter 4).

- I perform a simple *design space exploration* to determine a sensible implementation for a single tile of the Loki architecture (Chapter 5).

- I demonstrate that *efficient communication opens up a wide range of parallelism opportunities*, and that it becomes possible to map programs to the architecture in ways which are tailored to each program, or even each phase of an individual program (Chapter 6).
- I show that as well as improving performance, *mapping an application across multiple cores can be used to reduce energy consumption* (Chapter 6).
- I *explore software and hardware optimisations* to further increase the profitability of parallelism (Chapter 6).

My work was undertaken as part of the larger Loki project, and the design of the architecture was shaped by all members of the group and the needs of their research. I therefore must clarify which parts of this thesis are *not* my work. As far as possible, I treat others' contributions as constants; I may discuss opportunities allowed by their work, but do not explore them.

- The compilation toolchain was developed by Alex Bradbury. This had influences on the choice of instruction set, and on the communication mechanism between cores. Alex also performed an energy and performance characterisation of an ARM processor.
- The memory system from the L1 cache onwards is the work of Andreas Koltes. Again, the instruction set was influenced, and some small additions were made to the core's pipeline to optimise cache accesses.
- Robert Mullins made a significant contribution to the energy modelling framework. This included many base SystemVerilog designs, and semi-automation of the synthesis tools. He also produced the instruction set encoding.

1.3 Publication

A summary of this dissertation, with a particular focus on Chapters 3 and 6 is presented in the following peer reviewed publication.

D. Bates, A. Bradbury, A. Koltes, R. Mullins, *Exploiting tightly-coupled cores*, International Conference on Embedded Computer Systems: Architectures, Modeling, and Simulation (SAMOS XIII), July 2013

CHAPTER 2

BACKGROUND

This chapter covers the main features of the architectures which have influenced Loki's design in some way. It starts with a high-level overview of broad classes of computer architecture, and then drills down into work which is more-closely related to Loki. It is of course impossible to cover all influential work, but an attempt has been made to include everything significant.

2.1 History

This section gives a brief overview of the design of computer processors leading up to the present day. Reasons for design decisions are explained, and the problems facing today's computer architects are discussed. Much of the information is drawn from work by Horowitz [54] and Weste & Harris [133].

The first digital computers were necessarily very simple; they were limited by the designers' ability to manually draw and place all of the components in the design phase and maintain the finished machine. Early machines were also very large, expensive and power-hungry.

The invention of the transistor greatly improved the size and power consumption of early computers, but reliabililty was still an issue – the mean time to failure of the Harwell CADET was around 90 minutes [76]. Advances in transistor design and manufacturing technology provided steady improvements.

Integrated circuits in the 1970s brought about another step-change, allowing thousands of transistors to be placed on a single chip, each of which was much cheaper, much more reliable, much faster and much lower-power than its discrete counterpart. Continued refinement of the manufacturing process has continued to this day, with billions of transistors per chip now possible.

Performance of microprocessors increased exponentially at around 50% per year between the mid-1980s and early 2000s. This was due to advancements in several areas. The execution time of a program can be broken down to expose a number of factors which we are able to target to improve performance, and reduction of any one of these will improve the overall execution time:

$$\frac{time}{program} = \frac{time}{cycles} \times \frac{cycles}{instructions} \times \frac{instructions}{program}$$

The clock period of microprocessors ($\frac{time}{cycle}$) has improved greatly through a combination of advancements in manufacturing technology and pipelining.

In an attempt to keep up with the self-fulfilling prophecy that is Moore's Law, semi-conductor manufacturers have been steadily shrinking the sizes of features that can be implemented on sil-

icon chips. Each generation, the minimum feature size shrinks by 30%, and smaller transistors hold less electric charge, so are able to switch states about 1.4× quicker each generation. There were seven process generations in the two decades described, giving a 10× speedup. This trend looks set to continue for a while, at least, with many companies using Moore's Law as a roadmap and investing increasing sums in trying to maintain the exponential rate of development. However, we are nearing the end of CMOS, where statistics and quantum effects will mean that the number of dopant atoms in transistors will vary widely, resulting in different behaviours and even malfunction of some transistors.

In addition to improving existing transistors, new, faster logic families were discovered, offering roughly 2.5× improvements. Improvements to these low-level building blocks of digital circuits have wide-reaching effects across the design.

The advent of reduced instruction set computing (RISC) made pipelining much more achievable because of its simpler instructions with predictable execution times. Pipeline lengths increased, reducing the amount of logic between pipeline registers and allowing clock frequencies to rise. RISC was also helped by continuous improvements to compiler technology and memory capacity, making the lower instruction density less of an issue. Pipelines increased from 5 to 20+ stages during this period, allowing a 4× improvement in clock rates. RISC was so successful in driving clock frequencies up that many architectures with more-complex instruction sets started to internally translate their instructions into RISC-like microcode so that they could take advantage of the same techniques. This trend was not sustainable for two main reasons:

1. Increasing the clock frequency increased the power consumed by the chip superlinearly, as splitting the pipeline into more stages required complex logic and additional pipeline registers, and each transistor was switching on and off more times per second [111]. The power dissipation of processors hit various limits depending on the application domain – the *power wall*. Mobile devices such as laptops and mobile phones became more popular, and had very strict energy constraints imposed by their limited battery supplies. In the desktop and server market, the limits of how much heat could be cheaply dissipated from the chip were being reached. Servers have now reached a point where the cost of powering and cooling the chip over its lifetime is more than the cost of the chip itself [26].

2. There are not many natural points to split the pipeline up. As the number of stages increased, more of the fundamental pieces of the pipeline, such as the ALU and cache, needed to be spread across multiple stages. This increased complexity, and was not always worthwhile due to non-linear performance gains [111].

Today, with power consumption as a first-order design constraint, pipeline lengths have dropped and there has even been convergence between the mobile and desktop worlds: Intel's Core i series has 14-16 stages (the precise number has not been revealed) [38], and ARM's A15 has 15 stages [74].

The process of shrinking transistors not only improved their performance and energy consumption, but also allowed more of them to fit on a single chip. This paved the way for additional logic and memory, which allow more work to be performed each cycle (reducing $\frac{cycles}{instruction}$, otherwise known as CPI).

The structure of pipelines changed so that more work could be done at once. Datapath widths increased from 8 to 64 bits, allowing manipulation of larger numbers with fewer instructions. There are limits to how far this trend can continue because increasing the bus width adds an overhead to any instructions which do not need the full width, and there is currently limited need for anything beyond 64 bits. Many architectures now offer special vector instructions and

registers, allowing the same operation to be performed on multiple operands simultaneously without impacting the common case.

Pipelines also introduced multiple functional units to allow multiple independent instructions to be executed simultaneously. Again, there are limits to the number of independent instructions which can be found. While there is often a large amount of available ILP [83], it is usually hard to access. Finding enough independent instructions to supply all functional units requires speculation across multiple branches and accesses to many separate parts of the instruction (and data) store simultaneously, which is expensive to support.

Compiler technology improved greatly as higher-level languages became more popular, reducing the number of instructions used to execute a program.

In total, these changes to caches, datapaths and compilers delivered a 5-8$\times$ improvement in SPECint/MHz, a measure of how productive a processor is each clock cycle. This improvement took place even with the much-reduced clock period.

In recent years, to try to continue improving performance of microprocessors despite all of the obstacles in the way, chip designers have turned to placing multiple cores on each chip. The performance of each core now increases only slowly, if at all, and instead the number of cores on each chip increases exponentially. Mobile devices with four or more cores are not uncommon today.

The problem with this new approach is that in order to take advantage of the multiple cores, there needs to be available thread level parallelism (TLP). It is not too difficult to find small amounts of TLP: multiple programs can run at the same time, or a single program can have separate threads for the user interface and the background processing to keep everything running smoothly, but TLP is not scaling anywhere near as quickly as the number of cores per chip, despite the decades of research that have been put into parallel computation [21].

The work in this dissertation looks ahead a few years, assuming that the number of cores keeps increasing, to a time where single chips can have hundreds or thousands of cores. I explore how these cores might be implemented and how such a large number of them can be managed and utilised effectively. Available thread level parallelism is expected to rise slightly, as devices perform an increasing amount of background processing, and programmers slowly begin to write more-parallel applications, but I concentrate on exploiting other forms of parallelism within individual applications to make use of the many cores.

2.2 Classes of architecture

This section describes the main broad classes of architecture which currently exist, and the tradeoffs between them. Each class is designed for a different purpose and is at a different point in the design space encompassing programmability, flexibility, cost and energy efficiency. There are also architectures which attempt to bridge the gaps between different classes to combine a number of different desirable traits.

2.2.1 Multi-cores

Multi-core processors are ubiquitous today, in devices ranging from mobile phones to servers. They are an evolution of the "standard" single-core chips which came previously; they consist of multiple cores with additional logic to ensure that shared structures (such as caches and memory contollers) are always in a consistent state. Architects were forced to head in this direction because of the diminishing returns in making single cores faster and more complex.

Many multi-cores are homogeneous (Intel, AMD, Tilera), but some are heterogeneous to optimise for the various use cases. Cell [46] offers one complex core for handing control-intensive code, and eight simpler vector units optimised to speed up data processing. Modern mobile phone processors typically have two to four general purpose cores, and a large number of accelerators to improve performance and reduce energy consumption of common operations such as video decoding.

General-purpose processors are the easiest of these architectures to program. There is an established compiler infrastructure which can transform a large range of high-level languages to the required machine code. The problems are that writing code for multiple cores is often difficult due to the need to find parallelism whilst avoiding conflicts and deadlock, and that the flexibility introduces inefficiency. Only a small portion of the area of a processor is devoted to computation; the rest is there to make sure there is a steady supply of instructions and data, and that the instructions can execute quickly. It has been shown that only 6% of the energy of a representative general-purpose processor is used to perform arithmetic [33].

2.2.2 Graphical processing units (GPUs)

With the increasing importance of graphical user interfaces and smooth 3D graphics in games, architectures specialised for computer graphics have become common. Typical workloads involve massive data-level parallelism, since graphical computations often involve performing the same task for each pixel on the screen or each vertex in a model.

The usual approach is to have a huge number of threads. These are organised into *thread blocks* which represent the computation to be performed, and are in turn split into *warps* (in NVIDIA's nomenclature). A warp may consist of 32 threads, all of which execute the same code simultaneously on one of the GPU's *streaming multiprocessors*. This means that only one instruction fetch unit is required for each warp, greatly reducing the resources spent supplying instructions to the functional units. The huge number of threads is used to hide memory latency: if a warp requests data which is not held locally, instructions from other warps are issued until the data arrives. This reduces the need for on-chip cache, making space for more functional units.

GPUs are more energy efficient and perform better than general-purpose processors for their domain of applications, but their specialisation means that they are useful less of the time. When the number of threads drops, performance is impacted since it is no longer possible to hide memory latency. Dealing with control flow is also awkward. Since all threads in a warp execute the same instruction at the same time, there are problems if different threads are on different paths. In practice, all required paths are executed sequentially, with some way of masking out threads which do not need to execute the current instructions.

There is a movement towards making GPUs more general purpose and easier to program so that their efficiency can be used more of the time [95], and a similar movement towards making it possible for general purpose processors to exploit increasing amounts of data-level parallelism [1]. Taking one step further, OpenCL [67] provides a framework for writing parallel code which can be used across both CPUs and GPUs. This trend also includes hardware modifications such as adding increasingly-wide SIMD units to general-purpose processors, and adding scalar execution resources to GPUs [85].

The two classes of architecture are also physically being brought closer together, with many companies providing general-purpose processors and a number of graphics cores on the same chip. In addition, AMD plans to introduce a shared ISA and dynamic run-time compilation, so

that code can transparently migrate between GPU and CPU depending on the available parallelism [110]. This trend suggests that perhaps in the future the two architectures will converge as a massively-parallel, general-purpose architecture.

2.2.3 Field-programmable gate arrays (FPGAs)

FPGAs consist of a large number of lookup tables (LUTs) and a configurable interconnect which can be used to simulate arbitrary digital circuits. Each logic block is a small memory which is addressed using a number of input lines, and outputs a result.

A simple example of a LUT has two input lines (a two-bit address space) and a single bit of output. It therefore needs 2^2 entries of one bit each. Depending on the values of the input lines, different entries are read, and so various logic gates can be simulated. For example, setting entry 3 (11_2) to 1, and all other entries to 0 makes the LUT act like an AND gate, where an output of 1 is only produced if both inputs are 1 (Table 2.1). LUTs of typical modern FPGAs are more complex than this to reduce the overheads of routing signals between them.

In_0	In_1	LUT contents
0	0	0
0	1	0
1	0	0
1	1	1

Table 2.1: FPGA lookup table emulating an AND gate.

The first commercial FPGA, in 1985, had 128 LUTs, each with three inputs. A modern FPGA, such as Xilinx's Virtex 7, can have hundreds of thousands of LUTs, each with six inputs [135].

FPGAs are very flexible and can be reprogrammed to accelerate different applications, but the overheads of their configurability mean that they are not very efficient implementation platforms for general purpose code. They are good for low-level bit manipulation, because the datapaths are only a few bits wide, but combining multiple blocks to form wider datapaths introduces inefficiencies. They are also difficult to program – since they are designed for digital circuit simulation, they need to be programmed in low-level hardware description languages (HDLs). Tools exist to convert from high level languages to HDLs [94], but they have not yet reached the mainstream.

A study has shown that FPGAs require an average of 40 times more area, 12 times more power and 3 times more time to execute a program than a comparable ASIC [73]. Modern FPGAs now contain specialised hard blocks to reduce this difference; multipliers, block memories, and even entire processor cores can be found on FPGA chips.

2.2.4 Application-specific integrated circuits (ASICs)

ASICs are circuits which have been specialised for a particular application or domain of applications. For example, mobile phones may contain ASICs to accelerate digital signal processing and image processing. ASICs are generally the most efficient implementation of an application, and achieve the best performance and lowest energy consumption. They are able to tune all of the datapaths and storage elements so they are exactly as large as required (minimising wasted

space) and optimise the placement of subcomponents to minimise communication distances (reducing energy and time) [51].

The problem with ASICs is that they are expensive, and their cost is rising exponentially [104, 116]. Designing an ASIC requires experts in both hardware design and the target application area of the ASIC, and engineering costs of producing new chips are rising due to the increasingly complex processes involved with modern technologies. It is only economically viable to produce an ASIC if many million will be produced (6.5 million at 28nm [116]), and this reduces the number of applications they can feasibly be used to accelerate because they are often not cost-effective to produce.

In practice, ASICs often have some degree of flexibility so they can be used in more situations and justify the engineering costs. For example, a video decoding unit may also be capable of encoding the video, and of processing static images.

2.2.5 Comparison with Loki

In advance of a full description of the Loki architecture (Chapter 3), a comparison is made with each of these major classes of architecture to show where Loki fits in to the overall design landscape. Loki aims to take the best features of all of these architectures, while leaving behind the negative aspects.

Loki's design is a lot like that of a general-purpose multi-core: each of the cores can independently run full programs which have been compiled from high-level languages using standard compiler toolchains. The aim is to make it easy to program for multiple cores by making communication very cheap (and therefore less of an obstacle) and by exposing the interconnect to the compiler and programmer. The ability to bypass unused functionality is provided whenever possible to reduce energy consumption.

There are similarities with GPUs because of the large number of very simple cores and the sorts of parallelism Loki is able to exploit. Loki is easier to program, because of its support for high-level languages, and more flexible, because it is able to exploit more forms of parallelism. Loki could be seen as a potential result of the observed CPU-GPU convergence.

Loki could also be seen as similar to an early FPGA, but instead of configurable logic blocks, Loki has entire processing cores. Cores are generally not used alone (like LUTs), but are used as a platform upon which a software *overlay* can be mapped, so that the architecture is tuned to the application. This concept is explored in Chapter 6. Each core in the overlay is specialised by specifying the software it will execute, rather than statically storing data to a block of memory. This software may deactivate unneeded parts of the pipeline, or ensure that all necessary data and instructions are stored locally and can be accessed cheaply. The mapping can be changed rapidly if the structure of the program changes, or if a different program is to be executed. While FPGAs are useful targets for mapping low level digital circuits, Loki is a useful target for mapping high-level software.

Loki is similar to an ASIC in that it discards any functionality which is not needed in order to create an optimised platform on which a program can be executed. In Loki's case, the functionality is discarded by software at runtime, rather than by the architect at design time. The unused components remain on the chip, increasing the distances and communication costs between the components which are in use, but there is no switching activity within them and techniques such as clock gating can be applied so they consume virtually no energy. Loki is flexible enough that it can be designed and validated once, and then used for a wide range of applications, reducing costs.

2.3 Related work

In the last decade or so, there have been many architectures which may look similar to Loki at first glance, but they have a variety of different goals. This section describes a selection of them.

2.3.1 Network

With the need to decentralise resources to reduce communication distances and costs, some form of interconnect is required to transfer information between distant components. This section discusses some architectures which treat their communication networks as main components, rather than peripherals.

The **transputer** [2] in the 1980s was built around the idea that there should be a small number of different types of processor which are used for all purposes, from central processing units to disk drive controllers. Each transputer aimed to be cheap, but could be combined with other transputers to increase performance. Spare cycles of nearby idle transputers could be "borrowed" to increase performance. The transputer was designed to perform parallel computations: serial links were provided to up to four neighbouring transputers which allowed very fast communication. Processes communicated via channels, which were mapped to a set of reserved memory addresses [56]. This allowed a uniform interface between processes, whether they were on the same transputer or not. The occam parallel programming language was used to expose the parallelism to the programmer (though more-traditional languages were also compatible).

The transputer saw limited commercial success, as the exponentially increasing performance of traditional architectures meant that by the time the transputer was released, it was not significantly faster than competing sequential processors and so it was often not worth porting programs to the parallel transputer. It was also difficult to achieve the target cost. The design did find its niche in real-time and massively parallel computing, however, and is still used in set-top boxes from STMicroelectronics [3].

The **IBM Victor** family of workstations took advantage of the transputer's available serial links to create a two-dimensional grid structure and explore highly parallel message passing hardware [119]. Up to 256 transputers could be linked together and popular applications included parallel database management. The grid could be partitioned into arbitrary shapes, allowing users to execute independent applications without interfering with each other.

The structure of **Raw** [124, 125], one of the earliest single-chip architectures in this field, looks a lot like that of Victor. Raw also had a two-dimensional *scalar operand network* and popularised the ability for cores to send data directly to each other, instead of having to go through the memory hierarchy. Each core is paired with a static router and two dynamic routers to form a *tile*, and a program consists of code for the core and for the static router. Static routers are used to communicate operands, while the dynamic routers are used for unpredictable events such as cache misses. Raw exploits instruction-level and streaming parallelism on its tightly-coupled cores with some good results. The Raw group define a 5-tuple to allow comparison between different networks, consisting of:

1. Number of cycles wasted by ALU sending an operand
2. Number of cycles where ALU isn't wasted before message is sent
3. Average hop latency, in cycles, between adjacent ALUs

4. Number of cycles between operand arriving and being ready to use

5. Number of cycles wasted by ALU receiving an operand

A superscalar processor has a perfect 5-tuple of <0,0,0,0,0> as operands can be used as soon as they have been computed. A multiprocessor typically has tuple components which sum to hundreds of cycles, since communications must pass through multiple levels of the memory hierarchy. Raw achieves <0,0,1,2,0> by integrating communication into the instruction set and Loki improves this further to <0,0,1,0,0> by connecting the network directly to the pipeline – the only communication latency being a single cycle hop over the network. Section 6.4.6 explores the possibility of achieving the perfect 5-tuple on Loki.

The Raw architecture was later commercialised as **Tilera** [127, 132]. Tilera's Tile processors refine Raw's static network by adding an auxiliary processor to each tile to aid network reconfiguration, and add further dynamic networks. The observation is made that wiring is almost free, as it makes use of the highest metal layers on the chip, and is therefore above most of the logic and memory. Additional networks therefore add little overhead. An interesting feature of the Tile processors is that network buffers behave like a cache of streams between cores: if data from a new stream arrives, one buffer is selected to have its contents spilled to memory before being reallocated to the new stream.

XMOS [86] refined the concept of a network channel by providing a hardware mapping between logical and physical addresses, and allowing channel addresses to be communicated between cores in the same way as any other data. Links are one byte wide and it is possible to set up virtual circuits or use dynamic packet routing.

Like many of the above architectures, Loki attempts to reduce the overheads of parallelism by making the network more accessible to software. In Loki's case, this is done at the instruction set level, with input buffers mapped to registers and most instructions being able to specify a network destination for their results. Loki provides a rich interconnect structure with multiple subnetworks optimised for different purposes: a fast, local, point-to-point network; buses capable of multicasting values and a higher-latency network for less-frequent, longer-distance communications. All networks are 32 bits wide and are capable of transmitting instructions, data and network addresses.

2.3.2 Instruction supply

When making use of a distributed system, such as an architecture with an on-chip network, it is often useful to assign relatively large blocks of instructions to a core at once, to minimise communication required with shared, centralised structures. On Loki, these are called *instruction packets* (IPKs), and they roughly correspond to basic blocks in the program. This concept has been exploited in different ways by different groups, both to improve performance and reduce power consumption.

Elm [14] is a tiled, many-core architecture optimised for low energy consumption. Optimisations explored are mostly within the pipeline: deep storage hierarchies are provided and exposed to software to allow maximum exploitation of temporal and spatial locality. Elm also provides cheap, register-mapped communication between groups of four cores called an ensemble. Elm's instruction supply is completely compiler managed. The compiler requests a known number of instructions from a particular memory address, and places them in a fixed location in the core's instruction store. This allows very effective use of a small instruction cache, but can require instructions to be pessimistically re-fetched if the compiler cannot be sure whether they are already stored locally.

SCALE [72] introduced the concept of vector-threading, an abstraction which provides a control processor and a vector of virtual processors. Each virtual processor can execute independently, or they can join together to create a vector processor, with instructions fetched by the control processor, when data-level parallelism is available. Instructions are fetched in groups called atomic instruction blocks (corresponding to hyperblocks) to increase the amount of work received when a fetch is issued.

TRIPS [28] is a large project which aims to create a dataflow architecture by providing a large number of functional units in each core and having instructions specify destination instructions rather than destination registers, reducing the number of accesses to a large register file. Up to 128 instructions are bundled into *blocks*, which are dynamically scheduled across multiple dataflow processors, each with many functional units. Each block contains a hyperblock of instructions. TRIPS and a number of other architectures have demonstrated that scheduling instructions in this way allows locality to be exploited more effectively, which in turn allows a smaller and cheaper local storage structure to be used frequently [30, 106]. Executing instructions in atomic blocks means that only those values which are live before a block begins or after a block completes may need to be shared; all intermediate values within the block can be stored efficiently in a local structure. This simplification can also make detection of dependencies between blocks simpler, allowing multiple blocks to be executed in parallel.

Multiscalar [39] is able to group instructions together into tasks which range in granularity from a single basic block to an entire function call or more. Blocks are chosen to minimise dependencies between them, and then executed in parallel across a number of processing elements. Any remaining control dependencies between blocks are hidden by speculation, and data dependencies are resolved by communicating data between adjacent processing elements using a ring network. This execution style unifies a number of different ways of exploiting instruction-level parallelism, including software pipelining, task-level pipelines and dataflow.

Loki primarily uses instruction packets to reduce the amount of communication required between cores and memories, but is also able to distribute packets across cores to take advantage of efficient local storage. Loki's flexible inter-core communication structure means it is able to emulate the vector-processing used by SCALE and the dataflow pattern used by TRIPS. The contributions of this dissertation largely revolve around combining cores to improve performance or reduce energy consumption, and so are orthogonal to the pipeline optimisations of Elm.

2.3.3 Reconfiguration

Mai et al. recognised that different applications were best suited by different memory systems, and so designed **Smart Memories** with a degree of configurability built in [81, 82]. Each cache line contains four bits of metadata, and each memory bank contains a reconfigurable logic block which manipulates the metadata, and can itself be configured using special memory operations. Physical memory can be partitioned into multiple virtual memories, each with different capacities, line sizes, replacement policies, etc. These virtual structures allow *overlays* to be built which provide the best mix of resources for each individual application.

NVIDIA's research GPU architecture, **Echelon**, also allows the memory hierarchy to be configured to suit each application [65]. Each memory bank can be used either as a cache or a scratchpad, and has a malleable pointer to its parent bank(s), allowing a custom hierarchy to be created. Echelon is designed by many of the same people that worked on Elm, and

contains some of the same optimisations such as a hierarchy of register files to reduce energy consumption.

ACRES [9] was an early proposal for a tiled architecture. The contents of a tile were left unspecified, but a number of important challenges and opportunities were identified. ACRES supported both traditional memory-based instructions and state machines to control execution, and allowed a spectrum of execution patterns ranging from each core processing its own instruction stream to a data streaming mechanism where the results of one core were sent directly to the next. The authors also explored mapping virtual architectures across the homogeneous fabric, making use of configurable memory systems and compute resources. Furthermore, these virtual architectures could adapt dynamically to contention and available resources. Dynamic reconfiguration is discussed further in Section 2.3.4.

AsAP [128] has 164 very simple, 16 bit cores running at a high clock frequency, connected with a mesh network. Since each core has very limited memory resources (128×35 bit instruction memory and 128×16 bit data memory), ways of mapping applications in a streaming fashion are explored to reduce the need for temporary intermediate storage. Custom overlays are created for each program where each core has a very specific task and communication channels are set up statically. Each core's voltage and frequency can also be adjusted to match its computation requirements. Three accelerators are included on-chip to speed up fast Fourier transforms, motion detection and the Viterbi algorithm.

Loki allows customisation of both computation and storage at runtime. Typically this is done statically, but dynamic changes are also possible. Computation is configured by setting up a specialised communication network and optionally placing instructions and data on each core. Any core can communicate with any other, avoiding the problems of some more-restrictive networks where cores need to be spent simply to route data to its destination. The choice of instructions on each core affects which parts of the pipeline are used: an instruction is provided which bypasses cache tag checks when the location of an instruction packet is known statically, and when there is only one instruction on a core, much of the pipeline can be clock-gated. Loki cores are designed to be used together to exploit different forms of parallelism, depending on what is available in the program (or phase of the program). A number of these *execution patterns* are explored in Chapter 6.

Loki also has a configurable banked L1 cache in each tile; it is possible to set each bank as either a cache or scratchpad, and change its line-size and associativity. Multiple banks can also be grouped together to form virtual memories. Configuration is performed by sending special messages across the network to the memory banks, and can be done at runtime. Further details on the memory system are given in Section 3.4.

2.3.4 Dynamic configuration

It is well known that the resource requirements of an application are not always uniform throughout its lifetime. This can be exploited either by modifying the software to take advantage of the hardware resources available at the time (for example, scaling across more cores), or by reconfiguring the hardware to better match application needs.

Invasive computing is an example of the first approach [126]. Processes running on one or more cores are able to *infect* new cores to increase their compute resources or *retreat* to release resources. Distributions of processes across cores can change depending on computation requirements, resource availability, temperature, faultiness, permissions, etc. **PPA** [101] is an implementation of this. The instruction schedule generated by the compiler is virtualised so

that it can take advantage of varying numbers of available functional units. This works by using two bits in the encoding of each instruction to tell which core the instruction migrates to when resource allocations change.

Tartan also explores ways of spatially mapping programs to hardware [88]. Virtualisation is used to allow the program to continue running even when resources are limited. The problem is compared to both cache replacement (since sections of the program are swapped in and out of the available hardware) and circuit floorplanning (as communication costs between parts of the program should be minimised).

ARM have recently popularised their **big.LITTLE** technique to adapt the hardware at runtime [44]. This involves having two (or more) architecturally identical cores on a single chip, each targetted at different points in the performance/energy space. Applications are automatically moved between the cores depending on the performance they require. The voltage and clock frequency of each core can also be scaled to provide additional intermediate points.

Federation [25] and **Core Fusion** [59] both take multiple simple cores and merge them into single larger logical processors which are capable of exploiting various forms of parallelism. Federation starts with two in-order pipelines and focuses on forming an out-of-order dual-issue pipeline which expoits ILP to increase single-thread performance. Core Fusion starts with a dual-issue out-of-order design and focuses on joining resources such as caches and branch predictors of up to four cores to increase performance. Core Fusion in particular spawned much follow-up work in an attempt to reduce overheads and bring the performance of the fused core in line with that of a bespoke wide-issue processor [31, 91, 93, 106].

MorphCore [68] takes the opposite approach. The base architecture is a wide out-of-order processor, and the ability to exploit thread-level parallelism across the different lanes is added.

Rodrigues et al. instead provide a number of cores, each with a moderate capability to execute any program, but with a particular strength (for example, improved floating point performance) [109]. Interconnectivity between cores' pipelines allows cores to "trade" the weak parts of their pipelines with other cores' strong parts to improve performance or energy consumption, depending on application requirements. This technique is called **Dynamic Core Morphing**.

Similarly, Meng et al. describe a single instruction multiple data (SIMD) architecture called **Robust SIMD** which is able to provide a tradeoff between the width of the SIMD array and the number of threads which execute simultaneously [87].

Determining when the architecture should be reconfigured is also an active area of research. At one end of the spectrum are full-hardware approaches, where event counters and/or availability of nearby resources are monitored to determine whether reconfiguration would be beneficial. At the other end are full-software approaches where performance is measured by an operating system or periodic profiling phases. Software approaches have a higher overhead since they must interrupt the application's execution in order to perform their analysis, but are generally more flexible, and are able to try different heuristics more easily.

2.3.5 Miscellaneous

This section describes some specific techniques which have been used to achieve one or more of the goals that Loki has. In general, there is scope for applying these techniques to the Loki architecture, but this is not explored in this thesis.

Dynamic voltage and frequency scaling (**DVFS**) allows the voltage (energy) and frequency (performance) to be changed at runtime, depending on the needs of the application and the available resources. Two main applications of this technique are to put a core in a low-power

mode when its results are not needed urgently, or to temporarily boost power consumption above the TDP to compute a result quickly. Architectures such as AsAP (mentioned previously) have explored finer-grained versions of this technique which run cores which are not on the critical path as slowly as possible, so that the results are produced exactly when they are needed and with minimal energy consumption [128].

Computational sprinting [107] makes the observation that many applications are not characterised by a long sequence of sustained computation, but instead are made up of short sequences of intense activity separated by long periods of idleness. Since these bursts of activity are so short, it is often possible to increase the voltage and frequency (and therefore performance) to such an extent that the heat produced greatly exceeds the thermal and electrical capacities of the device, with the condition that there will be a recovery period afterwards. Several techniques are discussed which could increase the length, frequency and/or intensity of the bursts. **BubbleWrap** [64] takes this one step further. A large number of identical cores are placed on a chip, and the ones which are most power-efficient post-manufacture are selected to accelerate parallel code regions. All other cores are used individually at an elevated supply voltage and frequency to accelerate sequential code. The higher voltage significantly reduces each core's service life, and when the core fails, it is simply replaced with another. The elevated voltage is carefully chosen to maximise performance while reaching a target chip lifespan.

Core salvaging [105] observes that just because there is a permanent fault in a component, it does not mean that the component is useless. It may be the case that only a subset of functionality is lost, or that execution can still continue, but with reduced performance. Methods for classifying and adapting to such faults are explored.

StageNet [48] and **StageWeb** [47] provide extra interconnect in a multicore architecture so that faults only bring down individual pipeline stages and not entire cores. When a fault is found in a pipeline stage, the datapath is modified so that the same pipeline stage from another core is used instead, if one is available. Isolating faults at a finer granularity allows a larger number of faults to exist before the chip becomes unusable: in the best case it is possible to have one faulty pipeline stage in each core, but reroute the datapaths in such a way that only one pipeline is lost.

CGRA Express [103] and dynamic core morphing (DCM, mentioned previously) provide similar links between pipelines, but for aims other than increasing fault tolerance. This suggests that a rich interconnect structure can be used for a wide range of purposes. DCM's links allow cores to "trade" parts of their pipelines to increase performance of particular tasks or reduce power consumption. CGRA Express instead provides direct connections between neighbouring ALUs, and allows multiple operations to execute sequentially in a single cycle, reducing the latency of a sequence of operations.

Conservation cores [130] provides specialised yet configurable processors at design time to reduce energy consumption of frequently executed code. The specialisation inherent in these cores is used to address the utilisation wall. Since the processors are slightly configurable, they are resilient to updates to the programs being executed, thus extending the lifetime of the chip.

2.4 Homogeneity vs. heterogeneity

There is ongoing debate over whether the future of computer architecture will be homogeneous or heterogeneous. Advocates of homogeneity argue that a regular structure is simple, reducing design costs and making the architecture more scalable and easy to program. Those in favour

of heterogeneity suggest that specialised hardware is the only way to achieve the levels of performance and energy-efficiency demanded by modern applications.

Hill and Marty show that a dynamic homogeneous architecture is better than any static mix of performance- and throughput-targeted cores, because resources can be reallocated at run-time to suit the task being performed [53]. This work was later extended by Chung et al. to show that when specialised accelerators are added to the processor mix, they are the best option, suggesting that heterogeneity may in fact be best [32].

Loki lies firmly in the homogeneous camp: multiple identical cores and multiple identical cache banks form a tile, which is replicated across the chip. As well as the advantages of homogeneity described above, Loki uses its regular structure to implement fault tolerance: if a core does not work, execution can be re-mapped to another core. A similar process takes place at manufacture time for many modern commercial processors – some chips are simply marketed as having less functionality than they really have to keep yields high in the presence of faults [50]. Future work aims to allow Loki to cope with faults dynamically.

To address the concerns of the inefficiency of general-purpose, homogeneous architectures, Loki allows specialisation in software, bypassing unneeded functionality whenever possible. While this will not achieve the efficiency of dedicated hardware, it is believed to be a good compromise. Amdahl's law is addressed by allowing multiple cores to work together to accelerate sequential bottlenecks; Loki's low communication costs make this easier than in traditional architectures. Future work will also explore the impact of adding to each Loki tile specialised logic such as a shared floating point unit or configurable sub-graph accelerators for common operations.

2.5 Compilation

Compilation techniques are not part of this work, but a new compiler was required and has been implemented to make use of the new instruction set (described in Appendix A), and to take advantage of Loki's unique features. The compiler is based on clang and LLVM [75], so a wide range of analyses are supported.

Many of the features and optimisations described in this thesis could be utilised or performed by the compiler; potential algorithms are suggested throughout, but implementation is left for future work.

2.6 Future developments

As researchers attempt to maintain the rate of progress of computer systems, they will inevitably develop new technologies which will continue to change the landscape of tradeoffs and require further architectural changes. Examples of upcoming developments include: 3D chip stacking, allowing much more logic and/or cache, and much more efficient communication between the different layers; optical interconnect, allowing cheap communication over longer distances; new semiconductors which could provide a step-change in transistor switching speed or power consumption; and quantum computing, which will completely change the way computations are performed, making alternative algorithms and applications much more popular.

CHAPTER 3

LOKI ARCHITECTURE

In Norse mythology, Loki is a god capable of shape-shifting to suit the current situation. He is said to possess "tricks for every purpose", and when he wasn't causing mischief, he was liked by the other gods for providing useful tools which allowed them to complete their tasks more easily.

In this chapter I introduce the Loki architecture: a simple, homogeneous fabric of many cores, designed to allow flexible use of its resources. The Loki architecture is able to change its appearance by emulating other types of architecture which better suit the code being executed. I present an overview of the design and justify some of the more interesting design decisions. Energy and area models for each component are presented in Chapter 4, and the best implementation of each component is determined experimentally in Chapter 5.

Loki's novelty lies in its flexibility. While many other architectures are able to alternate between a small number of configurations, Loki's efficient, general-purpose communication network allows many more possibilities, and means that fewer applications have to settle for sub-optimal configurations.

Paradoxically, I believe that the difficulty of parallelising applications can be overcome by providing orders of magnitude more cores on a single chip than is typical today. In doing so, the cores must necessarily be very small, and so communication is often over a very short range, making it cheap and efficient. Loki's network is accessible directly through the instruction set to take advantage of this. Lifting the restrictions on inter-core communication mean that it could be easier to split a program into many tightly coupled "strands" than into a few mostly-independent threads.

Each core also consumes less energy, since the structures it accesses are smaller and data doesn't need to travel as far internally. This is useful in situations where performance is not the primary concern; execution can proceed slowly but cheaply on a single core. The design does not make use of long wires, which improves prospects for future scaling – logic scales better than wires when using modern fabrication processes.

The decision to have many relatively weak cores is consistent with Pollack's rule [23], which states that the performance of a microprocessor is roughly proportional to the square root of its area, suggesting that a large number of simple cores gives the best potential performance. Loki's individual cores are so simple that they do not perform as well as the more complex cores of other designs, but it is possible to fit so many more cores on a chip of the same area, that the total computation potential greatly increases. The challenge is then to find ways of effectively using the many cores to make up for the fact that each one is less powerful and address Amdahl's law. Section 5.5 compares Loki to a commercial embedded processor, and Chapter 6 describes ways in which the many cores can be used to realise this performance potential.

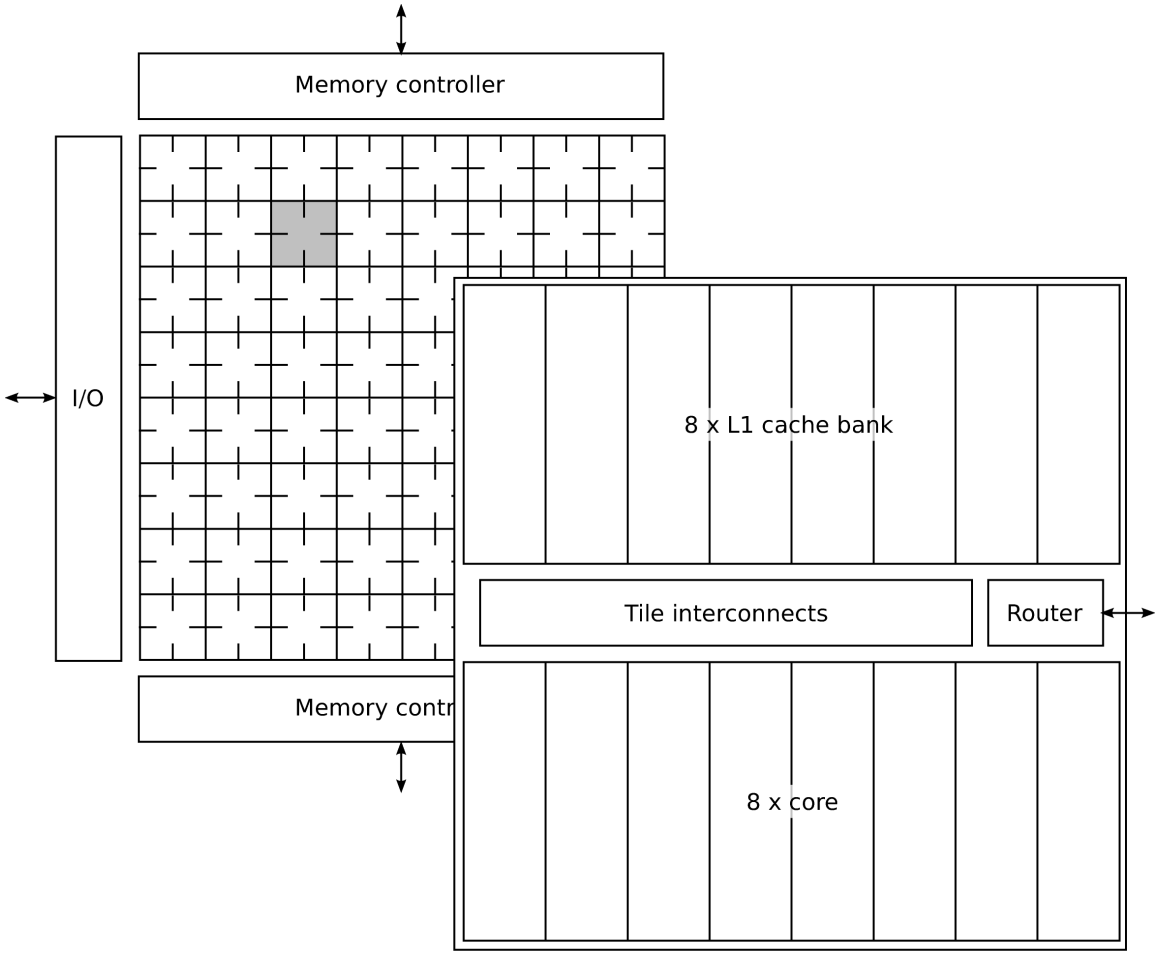


Figure 3.1: High-level Loki architecture, with chip (left) and one of its many tiles (right).

As discussed in Chapter 2, there has been much work in the area of low-power parallel systems, but none seem to address *all* of the issues computer architects will need to face in the near future.

3.1 Overview

Loki has a simple, tiled, homogeneous architecture (Figure 3.1). Each tile contains eight complete cores, each capable of executing its own instruction stream, eight 8kB memory banks, and various interconnect, and occupies approximately 1mm^2 in a 40nm process. (A more-detailed layout is provided in Section 5.4.) These tiles can be stamped out repeatedly across the chip, making scaling to large numbers of cores simple. The hierarchy of connectivity is optimised for the common case of local communication [43]; sending messages within a tile is cheap and efficient, and it is still possible for any component to communicate with any other if necessary.

The design can be seen as lying somewhere between that of an FPGA and a traditional multi-core. Although it is possible to execute any program on any individual core, we anticipate it being more common to map a program to a large number of cores, with caches and communication patterns set up differently in different parts of the mapping to optimise for the various features of the application. Unlike many other architectures in this area, Loki does not require a

separate control processor. The aim is to be flexible enough that cores can be composed in such a way that even sequential code can be sped up sufficiently.

In order to constrain Loki's design, I target embedded applications: the benchmark suite selected is for mobile devices, and I aim for a total power consumption of around 2W. Nevertheless, I expect that many of the findings in this thesis would translate to higher-power and higher-performance microarchitectures.

Loki was designed with three themes in mind:

- *Access*: making various structures accessible may allow novel use cases in unexpected situations. An inaccessible structure is one which only benefits a small number of instructions, or cannot easily be used to help another core. This philosophy also allows components to be reused more often, making the overall design smaller and simpler. The aim is to have as few solutions as possible for as many problems as possible – this is simply an application of the RISC philosophy to a forward-looking parallel architecture.

- *Bypass*: avoid using hardware components whenever possible to save energy (and perhaps improve performance). Examples include avoiding cache tag checks if it is already known that the instruction is stored locally, and using persistent instruction packets to remove the need for a jump in a long-running loop (Section 3.2.1).

- *Compose*: allow multiple cores to work together and share resources to execute a sequential thread faster. Multiple memory banks can also work together to provide the most-suited memory system for each application.

These features make Loki a very flexible platform – the aim was to be able to exploit the widest possible range of *execution patterns*, allowing virtual processors to be created which most closely match the natural structure of the programs being executed. An execution pattern is defined as any way of using multiple cores to execute a piece of code: this includes techniques such as using a core to prefetch or precompute data, and distributing independent loop iterations across multiple cores. Furthermore, I did not want the design of Loki to be constrained by the imaginations of those working on it; I wanted to be able to take advantage of new parallelism strategies without needing to change the hardware. This desire led to the target of making structures as general-purpose as possible, allowing them to be used in potentially unforeseen situations. Indeed, as the project progressed, I encountered many new ways for cores to cooperate which mapped naturally to the existing Loki architecture. Examples include deferring transaction code in software transactional memory systems to a helper core [80], and techniques for parallelising loops with cross-iteration dependencies [29]. Often, the effectiveness of these approaches is correlated with the latency of communicating between cores; I expect that they will generally perform well on Loki with its low-latency communication.

Lots of functionality is exposed to the compiler, including the network, delay slots and buffer contents. The architecture is sufficiently close to traditional designs that even the most basic of compilers should be able to generate functional code, but it should also be possible to use the extra information to optimise. For example, caches could be pre-filled with commonly-executed instructions, making it possible to know the relative locations of all instructions in a code section, and jump between them without requiring any cache tag checks.

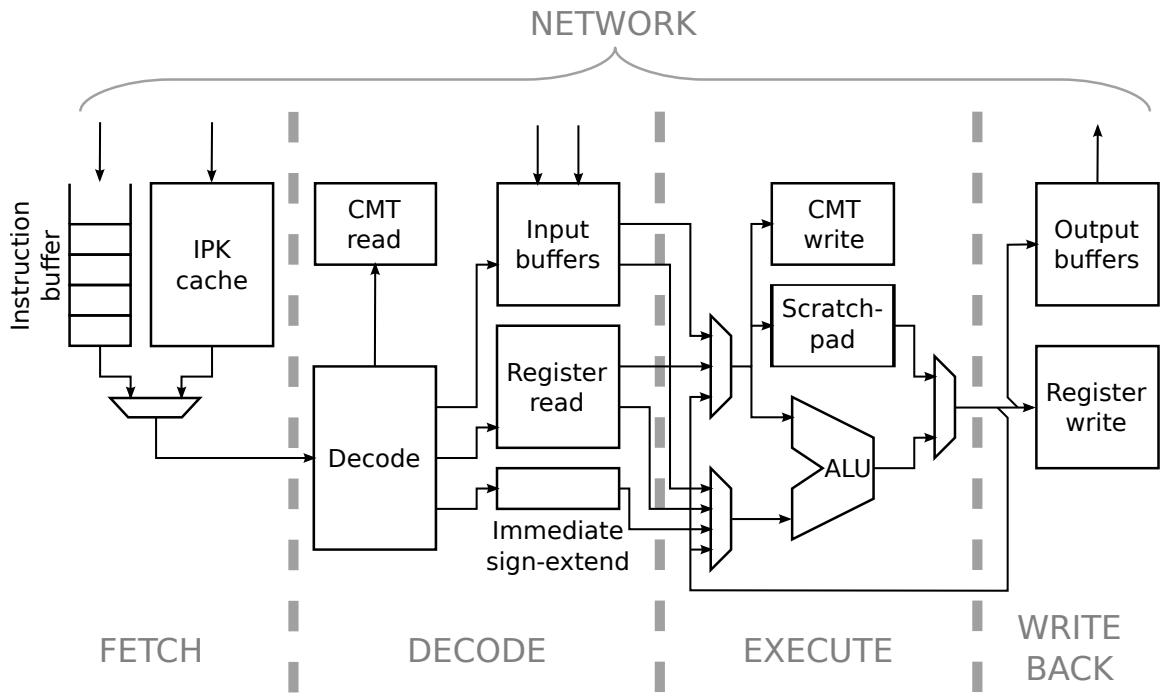


Figure 3.2: Loki pipeline block diagram.

3.2 Core microarchitecture

Decisions in computer architecture almost always involve tradeoffs. It is often possible to improve performance, for example, but this will likely have an adverse effect on energy consumption or area (or both). The Pareto front of the design space often has only a small region where a change in one metric will result in a proportional change in another [13]. Outside of this region, a large increase in energy consumption is required to effect a small increase in performance, and conversely, a small decrease in energy consumption requires a large decrease in performance.

For this reason, each Loki core is very simple, but not minimal (Figure 3.2). Loki aims to be at the energy-efficient end of the proportional region of the Pareto front. We therefore include features such as a shallow pipeline, a 32-bit datapath, and local storage for instructions and data, while eschewing features such as branch prediction, out-of-order execution and multiple-issue. We attempt to build everything out of a small number of basic building blocks, and make the various components accessible enough that they can be used for multiple purposes, when appropriate. This makes the cores smaller and lower-power (and in turn allowing cheaper communication and more cores), simpler to design and validate, easier to protect against faults, and allows more effort to be put towards optimising each component.

A shallow 4-stage pipeline is used. This reduces the number of pipeline registers needed; Section 4.5 shows that pipeline registers consume a relatively large proportion of energy and area in the pipeline. The short pipeline also reduces the need for branch prediction. Loki instead makes use of instruction prefetching where possible, and falls back on using a single branch delay slot (like MIPS).

In designing the Loki architecture, I tried to determine what a processor pipeline would look like if it was designed to communicate with other components. To this end, network connections are brought right into the pipeline and exposed to the programmer and compiler. Data from the network can be accessed in exactly the same way as if it were in a register, and many instructions

have the option to send their result over the network as well as storing it locally. The network logically connects every component to every other, but physically, it is made up of multiple sub-networks, each tailored to a particular use case. The networks are described in more detail in Section 3.3.

To increase uniformity and flexibility, L1 memory banks are also accessed over the network. This allows cores to masquerade as memories, applying a transformation to memory addresses before accessing the banks themselves (e.g. virtual memory), and removes the need for an explicit pipeline stage for memory access. It also makes the memory banks easily accessible to multiple cores for data sharing, and allows a single core to access multiple banks, increasing available storage space. The design is not far removed from that of other banked, shared caches, such as the level-2 cache of the Niagara series of processors [71].

In order to reduce the impact of the network latency when accessing memory, arbitration is done in advance when possible – the total time required to access the 64kB banked L1 cache is two clock cycles. For cores, this involves requesting network resources in the execute stage, so as soon as the value enters the write back stage, the network is ready. For memory banks, network resources are requested in parallel with the data array access.

Channels are a fundamental design feature which allow components to communicate. Each core and memory has associated with it a number of channel-ends, to which it can read or write. Each channel connects a single source to one or more destinations, and can be reallocated using a single instruction. In the cores, the input channel-ends are mapped to registers – reading from one of these registers retrieves data from a network buffer. A short code example is given later in Figure 3.5. These network operations are also blocking: if a value has not yet arrived or the destination of a sent message has no buffer space, the core will block until it is available. This makes synchronisation simpler, and is used as a building block for a higher-level message passing abstraction for the programmer. Message passing is used in preference to cache-coherent shared memory to produce the necessary fine-grain control of communications to minimise overheads, although simple shared memory schemes can also be supported.

To provide a level of abstraction to hide the network, we make use of the channel map table. This table maps logical network addresses to physical ones, and this provides multiple benefits:

- Fewer bits are required in the instruction encoding to specify a network destination.
- Network addresses are not hard-coded into the program, so communications can be re-mapped at runtime. This would be useful to move execution away from a hardware fault, or towards one of the chip's I/O pins, or for "paging" parts of the program on and off the chip (similar to Tartan [88]).
- In the future, it may be possible to provide some sort of protection or segmentation for memory, since all memory accesses are over the network.

The pipeline has two instruction inputs: there is a cache for frequently-executed instructions, and a FIFO queue for instructions which are expected to execute only once. Typically, the cache is used for instructions which the core fetches for itself, and the queue behaves like an interrupt mechanism: another core can direct instructions to the queue, and these have priority. This feature is often used to allow cores to work together. More information on the instruction fetch mechanism is in Section 3.2.1.

The register file has 32×32-bit entries with two read ports and one write port. Two registers are read-only and hold special values, and a further six are mapped to input network buffers. A larger data storage array is provided in the form of the scratchpad in the execute stage. The

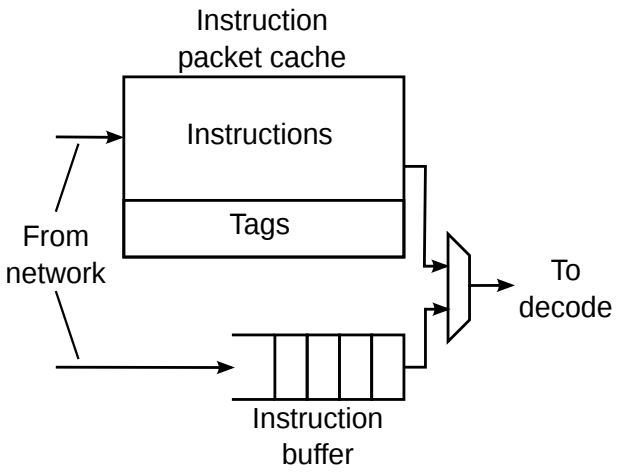


Figure 3.3: Loki instruction sources.

scratchpad is managed by software, and has a single read/write port. More detail can be found in Section 3.2.6.

3.2.1 Instruction fetch

Loki makes use of instruction packets (IPKs) to help decouple the memory banks from the cores, and to fit better with the network-centric design. Instructions are grouped into packets which roughly correspond to basic blocks: a single instruction fetch returns an entire packet, and once a packet begins execution, it is guaranteed to finish (unless cancelled in exceptional circumstances). The final instruction of each packet has an *end of packet* marker (in assembly, `.eop` is appended to the instruction name). Memory banks and caches autonomously continue reading instructions sequentially until this marker is seen.

The Loki pipeline contains two instruction inputs, as shown in Figure 3.3. One is a small FIFO queue which is used for instructions which are not expected to be needed again, such as requests from other cores to start executing a program, or sections of the program which do not exhibit good caching behaviour. The second is a small level-0 cache, referred to throughout this dissertation as the *L0 cache*, *instruction packet cache*, or *IPK cache*. The two instruction inputs were originally provided to allow a separate channel for cores to send instructions to each other. This is still a useful feature, but it has since been discovered that separate inputs also allow for some interesting optimisations (Section 5.1).

It is possible to switch between the cache and instruction buffer as the main instruction source at any time by sending a configuration message to the L1 cache and telling it where it should send instructions from now on. Further details on configuring memory are in Section 3.4.

Since Loki's L0 caches are so small (and L1 accesses are relatively expensive), superfluous instruction fetches are minimised by limiting each instruction packet to only a single exit point. This decision trades performance for energy efficiency: less spatial locality is exploited, but fewer unneeded instructions are fetched. The performance impact is reduced by prefetching instruction packets when possible. Having a single exit point per instruction packet means that packets correspond better to basic blocks than hyperblocks, though it is possible to put multiple basic blocks within a single instruction packet by making use of the *in-buffer jump* instruction. This instruction behaves like an unconditional branch to a relative position within the cache, and is used when the cache contents can be predicted statically. This instruction further reduces the

number of tag checks required, since it is already known that the instruction is at the specified location.

Level-0 caches have also found their uses in other architectures, and are typically used to reduce energy consumption. The Elm architecture has a compiler-managed L0 cache to reduce the costs of instruction supply [15], and Intel's Sandy Bridge processors contain a cache of decoded micro-operations which allows the first few pipeline stages to be bypassed about 80% of the time [57].

The use of instruction packets allows instructions to be prefetched easily, and also reduces the number of tag checks required in the L0 cache. Only the first instruction of the packet needs to be looked up, and all others can be read sequentially from there – the first instruction of each packet in the cache is automatically aligned with the next available cache tag. The cache's first-in-first-out replacement policy ensures that if the first instruction of a packet is in the cache, the rest of the packet will be there as well.

If necessary, fetch requests wait in the core's output buffer until there is space in the L0 cache for the entire packet. This ensures that the packet currently being executed is not overwritten, and that the source memory bank does not block part-way through delivering the packet, which could potentially lead to deadlock. Instead of specifying each instruction packet's length, a maximum size is defined, and it is pessimistically assumed that all packets are of this size. The maximum instruction packet size is typically half the size of the L0 cache. Software can ignore this restriction if it is known that the current packet will complete quickly enough that being overwritten is not a problem.

The cache is fully-associative with a FIFO replacement policy. This approach is used so that the cache stores a trace of instructions it has executed – this results in better utilisation of the limited resource because uncommon branches are not cached. A limitation is that if the body of a loop is larger than the available storage space, it will all need to be re-fetched on each iteration. Section 5.1.3 explores ways of reducing this problem. Another way of ensuring that cache space is used effectively is to hand control over to the compiler. Since the compiler is out of the scope of this work, this approach is not explored here, though instructions are provided to give the compiler partial control – these are described below. One problem with complete compiler control is that unpredictable control flow can result in limited knowledge of the contents of the cache, and so instructions may need to be pessimistically fetched from the next level of the memory hierarchy, even if they are already in the cache [102].

The instruction buffer has priority over the cache; if the core has completed execution of one instruction packet, and there are packets waiting in both the buffer and the cache, the packet from the buffer is chosen. This allows cores to interrupt each other, either to cancel the work they are doing, or to request a small amount of computation before returning to the previous instruction thread. Once a packet begins execution, it continues to completion, regardless of any other waiting instructions.

Loki offers multiple ways of managing the contents of the cache and buffer:

- *Fetch* instruction: request an instruction packet if it is not already cached locally. The packet is queued up to execute as soon as the current packet finishes, which means it can be prefetched well in advance to hide memory latency. There can be at most one *fetch* in progress at any one time, to avoid the need for deinterleaving instructions from different packets. There is a one-cycle branch delay slot when there is a hit in the L0 instruction cache: the instruction packet's memory address is computed (or read from a register) in the decode pipeline stage, and the L0 lookup is performed in the following cycle in the fetch stage.

- *Fill* instruction: request an instruction packet if it is not already cached locally, but do not execute it. This is useful if there are instruction packets which are known statically to be good to store in the cache. Once the cache has been filled with useful instructions, the contents can effectively be locked by directing all subsequent instructions to the instruction buffer instead. This allows an implementation of *cache pinning* [102], and is explored further in Section 5.1.3.

- Persistent instruction packets: these are instruction packets which execute repeatedly until either a *next instruction packet* command is received over the network, or a new packet is fetched. An instruction packet is made persistent by using the *fetch persistent* instruction in the place of an ordinary *fetch*. Each iteration of the instruction packet can be issued immediately after the previous one finishes; there is no bubble in the pipeline. When the core drops out of persistent mode, the pipeline is flushed, and any remaining instructions in the current iteration are cancelled. Persistent instruction packets are useful for implementing very tight loops, since the branch instruction is not needed. Predicated execution can be used to introduce simple control flow to a persistent packet. In the extreme case that the persistent packet contains a single instruction, much of the pipeline can be clock gated, and significant energy can be saved (Section 6.4).

- *In-buffer jump* instruction: if the contents of the cache are known statically, the core may jump around within its cache without the need for computing a memory address and checking all cache tags. This instruction has the advantage that it completes one pipeline stage earlier than the others, as no address computation needs to be performed. This means that there is no branch delay slot, and the next instruction can be issued in the following clock cycle.

For more details on the semantics of these instructions, see Appendix A.

The supply of instructions is completely software managed; there is no fall-through to the following instruction packet when a packet completes. This decision is made so that instructions are only fetched when they will definitely be used; this eliminates the energy costs of fetching unwanted instructions and prevents unnecessary evictions from the very small L0 cache. Elm, SCALE and TRIPS, which also make use of similar blocks of instructions, all do without fall-through and do not suffer a notable performance degradation [15, 28, 72].

A single instruction packet can execute at most one *fetch* instruction, but any number of *fill*s. This restricts the number of packets which can be queued up to one, and makes reasoning about the execution path easier.

Loki's ability to prefetch instruction packets means that the lack of branch prediction is not much of a penalty when executing code on a single core. When attempting to exploit instruction-level parallelism across multiple cores, however, speculation may be required to find enough independent instructions. Future work will explore whether it is possible to use additional cores to provide this functionality, or whether more-traditional prediction mechanisms are required.

3.2.2 Decode

The decoder is very simple thanks to Loki's RISC instruction set and simple encoding (Appendix A). Figure 3.4 shows the instruction formats used. For all formats which specify a particular field (e.g. a register index), the position and size of that field is the same, making it easy to extract the value quickly. The only exception to this is immediate values, which can be different sizes.

FF (fetch) format:

p	opcode	immediate
2	7	23

0R (zero registers) format:

p	opcode	xxxxx	channel	immediate
2	7	5	4	14

0Rnc (zero registers, no channel) format:

p	opcode	xxxxx	immediate
2	7	9	14

1R (one register) format:

p	opcode	reg1	channel	immediate
2	7	5	4	14

1Rnc (one register, no channel) format:

p	opcode	reg1	xx	immediate
2	7	5	2	16

2R (two registers) format:

p	opcode	reg1	channel	reg2	immediate
2	7	5	4	5	9

2Rnc (two registers, no channel) format:

p	opcode	reg1	xxxx	reg2	immediate
2	7	5	4	5	9

2Rs (two registers with shift amount) format:

p	opcode	reg1	channel	reg2	xxxx	immediate
2	7	5	4	5	4	5

3R (three registers) format:

p	opcode	reg1	channel	reg2	reg3	fn
2	7	5	4	5	5	4

Figure 3.4: Loki instruction formats. *p* represents the predicate bits, allowing execution to be conditional on the value of the predicate register, and marking the ends of instruction packets. *fn* is the ALU function. This encoding was produced by Robert Mullins.

Channel index	Hardware structure
0	Instruction buffer
1	Instruction packet cache
2-7	Registers 2-7

Table 3.1: Hardware components corresponding to each input channel of a Loki core.

After the instruction has been decoded, data is gathered from all appropriate sources: registers are read; network buffers are accessed, stalling the pipeline if data hasn't arrived yet; and the channel map table is accessed. Address computation for *fetch* instructions is also performed in this stage.

3.2.3 Execute

The execute stage contains an ALU capable of 15 functions and a two-cycle 32-bit multiplier. Scratchpad access and writes to the channel map table also take place in this stage.

3.2.4 Write back

The write back stage is very simple, performing register writes and sending data onto the network when permission is given.

3.2.5 Network integration

Each core has eight input channels and sixteen output channels. Each input channel is associated with a separate buffer, as described in Table 3.1, so that data from different sources can be accessed in a different order to the one in which they arrived. All output channels are multiplexed onto a single output buffer; it would also be possible to provide a buffer for each channel, or for each subnetwork (described in Section 3.3), but no significant advantage was observed in doing so. The final output channel (channel 15) is reserved as the *null channel*. It is used by instructions which have a space in their instruction encoding to specify an output channel, but do not want to send their result onto the network. Output channel 0 is implicitly used by all *fetch* and *fill* instructions to allow more bits in the instruction encoding to be allocated to an immediate value.

Most of the input channel-ends are mapped to registers – reading from one of these registers retrieves data from a network buffer. Reads are destructive; it is not possible to read the same item twice, so if this behaviour is desired, the value must first be moved to a register. Network operations are also blocking: if a value has not yet arrived or the destination of a send has no buffer space, the core will block until it is available. This makes synchronisation simpler, and can be used as a building block for a higher-level message passing abstraction for the programmer.

It is possible to read from two separate input channels in a single cycle, as though they were registers. The result of reading from the same channel twice in a single cycle is not defined, as it is not clear whether the same result should be returned both times, or whether two consecutive reads should be performed. Reading two consecutive values is often the more useful option, but allowing this to happen within a single clock cycle would greatly complicate the design of the network buffers.

To increase uniformity and flexibility, memory banks are also accessed over the network. This allows cores to masquerade as memories, applying a transformation to memory addresses before accessing the banks themselves. This could be useful for implementing virtual memory, memory protection, and transactional memory, for example. The network also makes the memory banks easily accessible to multiple cores. In order to reduce the impact of the network latency when accessing memory, arbitration is done in parallel with computation or memory access – the total time required to access the 64kB banked L1 cache is two clock cycles in a zero-load system.

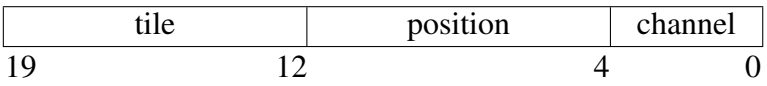
Each core contains a channel map table to translate between logical and physical network addresses. The table also keeps track of any flow control information such as the number of credits a channel has. This information can be used to determine whether the core is allowed to send more data onto a particular channel at a given time.

The channel map table is read in the decode stage (at the same time as the register file). If communicating with memory, the memory bank to be accessed is determined in the first fraction of the execute stage using the lowest bits of the target memory address, and a request for network resources is sent to the appropriate arbiter. If there are no competing requests, a grant will be issued in the same cycle. The data is sent onto the network at the beginning of the write back stage, and takes half a cycle to traverse the crossbar to the memory bank. The memory bank is clocked at the negative clock edge and has a one cycle latency, so is ready to send data back to the core on the following negative clock edge. After another half-cycle to traverse the crossbar, the data arrives at the core. Further detail about the timing of each component can be found in Section 4.5.

Channel address encoding

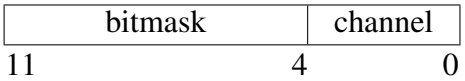
Network addresses are encoded as integers, allowing them to be manipulated by the ALU before they are stored in the channel map table, or shared between components. It is possible to specify point-to-point addresses to any component on the chip, and multicast addresses within a tile.

A point-to-point address is encoded as:



tile is the index of the tile on the chip; *position* is the component's position within the tile – cores range from 0 to 7 and memories range from 8 to 15; *channel* is the component's channel to be accessed – cores have 8 input channels and memories have 16.

Multicast addresses are encoded as:



bitmask is an 8-bit entry with one bit for each core in the local tile. The core at position 0 is represented by the least significant bit. Data sent to this address is to be sent to all cores which have their bits set in the bitmask. To simplify the encoding and reduce the amount of information required, a restriction is made that the data must be sent to the same input channel of all target cores. This restriction did not prove to be a limitation in any of the experiments which used multicast.

```
uint32_t updateCRC32(uint8_t ch, uint32_t crc)
{
  return crc_32_tab[(crc ^ ch) & 0xff] ^ (crc >> 8);
}
```

(a) C code

```
setchmapi 1, r11                # set up output channel 1
[...]
fetch     r10                   # pre-fetch next packet
xor       r11, r13, r14         # r11 = arg1 ^ arg2
lli       r12, %lo(crc_32_tab)  # lower 16 bits of label
lui       r12, %hi(crc_32_tab)  # upper 16 bits of label
andi      r11, r11, 255         # r11 = r11 & 255
slli      r11, r11, 2           # r11 = r11 << 2
addu      r11, r12, r11         # r11 = r12 + r11
ldw       0(r11) -> 1           # request data from memory
srli      r12, r14, 8           # r12 = r14 >> 8
xor.eop   r11, r2, r12          # use loaded data (r2)
```

(b) Loki assembly code

Figure 3.5: CRC code example showing how Loki's instruction set interacts with the on-chip network.

Example

Figure 3.5 lists a fragment of the kernel of the CRC benchmark. Before the kernel begins, *setchmapi* (*set channel map with immediate*) associates the logical network address `1` with the physical network address held in `r11`. The function itself begins with an instruction *fetch*: the next instruction packet to be executed is known immediately, and is fetched in advance. The load instruction (*ldw*) demonstrates the ability to send data onto the network with the `->` notation; most instructions are able to store their results locally, send them over the network, or both. The load works by sending a memory address over the network to the appropriate cache bank. The cache bank also has a channel map table which has been configured to send data back to channel 2 of the core when a request is received at a particular input channel. This data is used in the final instruction: registers 2-7 are mapped to the input buffers. The `.eop` marker denotes the end of the instruction packet and triggers the start of execution of the packet fetched earlier.

3.2.6 Data supply

Figure 3.6 shows how Loki's register file is arranged. The register file has 32-bit entries accessible through two read ports and one write port, all of which are visible in the instruction set. Register 0 is hardwired to zero; register 1 contains the address of the current instruction packet (the closest thing Loki has to a program counter) so relative jumps can be made; registers 2-7 are mapped to the input network buffers (excluding instruction inputs); and registers 8-31 are normal registers.

Scratchpad access requires special instructions and is demonstrated in Figure 3.7. The first two instructions store the constants 42 and 1000 in registers 10 and 11, respectively. The third

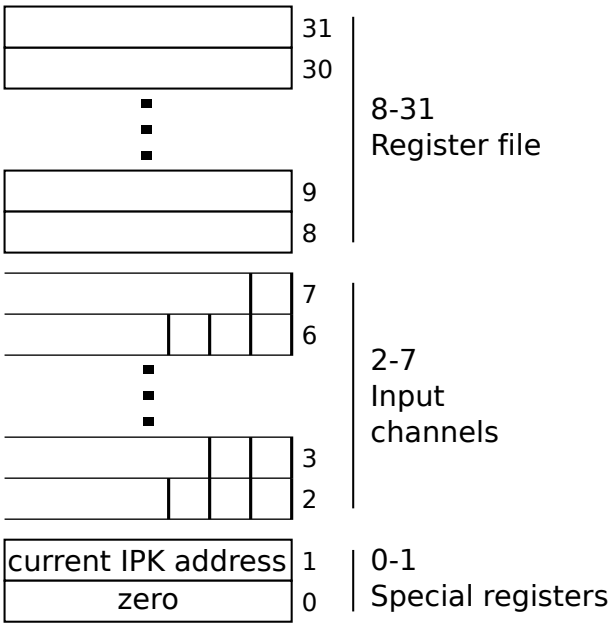


Figure 3.6: Register file.

```
lli r10, 42
lli r11, 1000
scratchwr r10, r11
scratchrd r12, r10
```

Figure 3.7: Loki assembly code demonstrating use of the scratchpad.

instruction (*scratchwr*) copies the contents of register 11 into the scratchpad position referenced by register 10. The final instruction (*scratchrd*) reads indirectly from `r10` and stores the result in `r12`. Since register 10 still holds the value 42, scratchpad index 42 is read, and the value 1000 is stored in register 12. There also exist immediate versions of these instructions, allowing the compiler (or programmer) to directly access particular entries of the scratchpad, and instructions which behave in the same way but address the input buffers (*iwr* and *ird*).

Indirect access allows better use of the input network channels when combined with the *select channel* instruction (*selch*). *Select channel* stores the index of a non-empty input channel to a specified register, stalling the pipeline until data arrives if all input channels are empty. An *indirect read* can then be performed to read from this channel. This behaviour is useful when many different components need to send data to a single core, and the core may process the data in any order.

All sensible optimisations to make use of the scratchpad will reduce energy consumption: it is much cheaper to access than the L1 cache, even if an extra instruction or two are required. Not all such optimisations will improve performance, however. In some cases, additional instructions are required to determine where in the scratchpad the required data is (or even if it is there at all), and these may negate any performance benefits gained by storing the data locally.

The scratchpad can be used for a variety of purposes including storing a small table of data, storing constants and branch target addresses, storing the contents of spilled registers, or storing a small number of stack frames.

When using the scratchpad to store a table, access through another register is most useful; as a space for storing constants, the compiler usually knows statically which entry is to be accessed, so using an immediate index is the better approach. One particular advantage of the scratchpad is its simplified addressing: element x of a table can be stored at position x in the scratchpad, eliminating the need to generate a memory address and reducing the number of instructions executed.

3.3 On-chip network

The network is central to Loki's design: communication between cores and memory access both take place over the network. If a cache suffers a miss, it also requests the new data over the network. Logically, all cores and memory banks communicate over the same network, but in practice, it is made up of multiple physical networks, each optimised for a particular communication pattern.

Each core and each memory bank has a network address, and the network to use is determined by very simple logic applied to the source and destination addresses. As well as specifying the component to send data to, the input channel of that component also needs to be given. Each input channel can have at most one writer: this simplifies arbitration. The writing component can change an arbitrary number of times during a program's execution, but there can be at most one at any point. This is enforced at the software level.

The network's design also reflects an attempt to avoid the possibility of deadlock, as deadlock recovery mechanisms are traditionally excessively complex and inelegant. The Raw architecture, for example, provided a duplicate, more-restricted network used for draining buffer contents to memory when its first network experienced deadlock [124]. Loki's network makes use of end-to-end flow control, so components only put data onto the network if it is guaranteed to be removed again. This avoids congestion, which in turn eliminates the risk of deadlock [52]. Non-blocking networks such as crossbars are used so that network traffic from different compo-

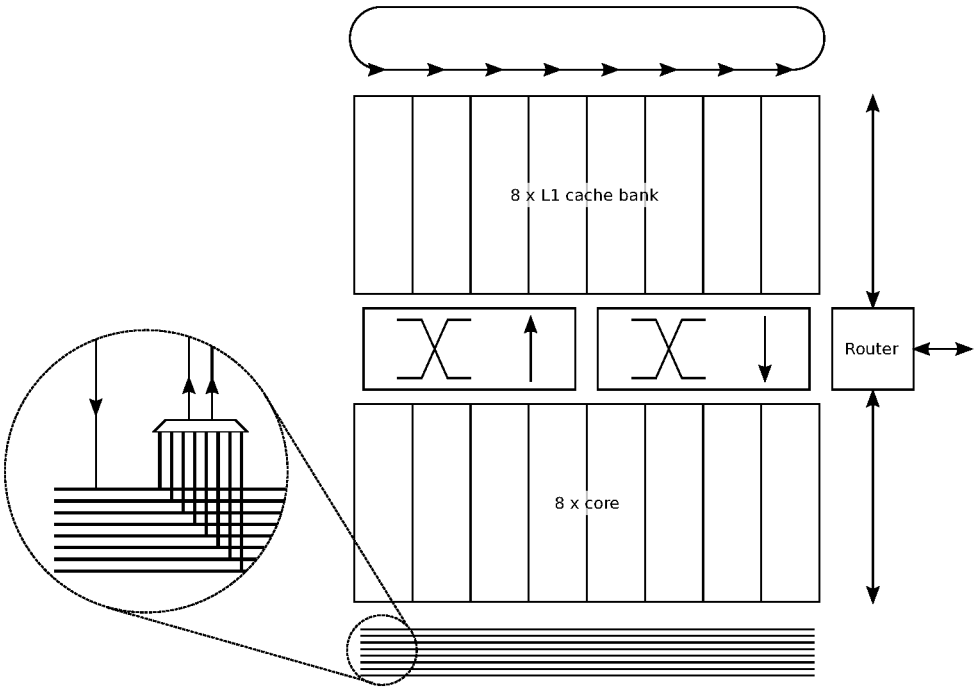


Figure 3.8: Sub-networks within each Loki tile. Cores and memory banks communicate with each other via fast crossbars, cores communicate with each other using buses capable of multicast, and memory banks communicate with each other using a simple ring network. A router provides links to neighbouring tiles.

nents doesn't interact in any way. (A non-blocking network is one in which an unused input to the network can always be connected to an unused output, without disrupting any existing communications.) Of course, deadlock is still possible if there is a cyclic dependency in software, but this would be unavoidable for such malformed programs.

It is expected that chip I/O and memory controllers will also be accessed through the network, as with Tilera's chips [132].

3.3.1 Intratile networks

The networks within a tile are optimised to be low-latency and low-energy. Communication to a nearby component is more common than to a distant one [43], so the network is optimised for this case. The cheaper network communication is, the more often it can be used profitably, and so the more opportunities there are to use communication to group multiple cores and memories together. To further reduce latency, arbitration is carried out in the cycle before data is due to be sent. In memory banks, this happens in parallel with accessing the SRAM, and in cores, it takes place in the execute stage. Assuming that the required network resources can be allocated, the data can then be sent as soon as it becomes available. If the required network resources are in use, the information is buffered until they become available.

Figure 3.8 shows how multiple sub-networks are used for different communication patterns. Between cores and memories there is a separate crossbar for communication in each direction, allowing a high bandwidth and low-latency connection to multiple memory banks. Using a non-blocking network such as a crossbar is important to prevent the many components from interfering with each other and potentially causing deadlock. After arbitration has been done, it is possible for data to traverse the crossbar in half a clock cycle. This helps reduce memory

latency. In total, in a zero-load system, the time required to access memory is two clock cycles: half a cycle to transmit the request to the memory bank, one cycle to access the memory bank (memory banks are clocked on the negative edge), and half a cycle to transmit the data back.

Cores communicate with each other over a different sub-network. Each core has a bus which it can write to, and it can specify that any subset of the cores in the same tile should receive the data. The same input channel index of each destination core is used to simplify the network address. Although this is also a non-blocking crossbar structure, I typically refer to the core-to-core network as the *multicast network* to distinguish it from the point-to-point communication used between cores and memory banks. This multicast capability helps several execution patterns (see Chapter 6), but the increased wire length and load increases energy costs and slows down data transmission so it takes a whole clock cycle. This is the reason why we do not use a multicast network to send from memory banks to cores, even though it may sometimes be useful; the extra latency was experimentally found to reduce performance by 9%. If data from a memory bank needs to be sent to multiple cores, it is instead requested by a single core, and then sent out on the core-to-core network, at a cost of one extra instruction and an additional network transfer.

Memory banks communicate with each other over a simple ring network, allowing streaming operations such as instruction packet reads to be handed on to the next bank if they extend beyond the contents of the current bank. Memory banks also share channel map table information on the ring network to facilitate this.

Finally, all components share a bus connection to the local router, which in turn is connected to the four neighbouring tiles.

Each core is capable of sending one word and receiving two words per cycle, from any combination of networks. In rare circumstances, a core may want to receive three words in one cycle (two operands and an instruction), but allowing this would require increased network complexity and be detrimental to the common case.

Memory banks are capable of sending and receiving one word per cycle to cores, and one transaction per cycle between banks. Commands received from other memory banks take priority as they include important operations such as updating the channel map table.

3.3.2 Intertile network

The work in this thesis concentrates on behaviour within a tile, so a placeholder intertile network is used.

It is expected that the network will be based on a simple mesh, where each tile has direct connections to its four immediate neighbours. This design is popular in networks-on-chip as it is scalable and keeps wire lengths low. Loki's clock period allows data to travel relatively large distances in a single cycle, potentially making the addition of longer-distance multi-drop channels useful to reduce the diameter of the network [45].

3.4 Memory hierarchy

The memory system is beyond the scope of this work, so only a brief overview is given here.

Each Loki tile contains eight memory banks, each with separate network access ports and command buffers. Each bank holds 8kB of data, and is accessible in a single clock cycle.

Banks are accessed by sending commands over the network. The source component computes which bank to access based on the lowest few bits of the address being accessed – cache

lines are interleaved between memory banks. A read command consists of a *read* opcode with a 32 bit address. A write command consists of two flits (flow control units): a *write* opcode with a 32 bit address, followed by a *payload* opcode with 32 bits of data. Wormhole routing is used throughout the on-chip network to ensure that these two flits can never interleave with memory commands from other components.

Memory banks must also know where to send data back to, since their only interface is to the network. Each memory bank has 16 channels, each associated with a separate return destination held in a table (a simpler version of the channel map table found in cores). All arriving memory commands are addressed to a particular channel, so if the memory operation produces a result, it will send it to the return address associated with that channel. The addresses can be changed at runtime using the *table update* command, allowing each core to have multiple connections to the same memory bank. This is useful because it allows instructions to be directed to either the L0 cache or instruction buffer of the target core, and it allows multiple data channels, allowing multiple loads to take place simultaneously and potentially complete out of order. Since the return address is known as soon as the operation begins, it is possible to request network resources before data is available – this helps reduce the overall memory access latency.

All memory banks in a tile are connected in a ring structure, as described in Section 3.3.1. If a memory bank does not contain the data requested of it, it sends a message onto the network to the level-2 cache, and blocks until the requested cache line arrives.

The proposed memory banks have some degree of configurability: the line-size and associativity of each bank can be configured, and it is possible to group neighbouring banks together to form larger virtual banks. It is also possible to switch cache tags off entirely and treat banks as software-managed scratchpads. This could be used, for example, to arrange instructions and data in memory such that each core mainly accesses the memory bank closest to it. This would reduce the lengths of wire toggled in the network, saving energy, and also reduce memory contention between cores. For all experiments in this thesis, a basic configuration was chosen with all eight banks merged into a single direct-mapped cache with cache lines of 32 bytes.

The level-2 memory system is left undefined for this work. It is modelled as a memory of unbounded size, with a latency of ten cycles. Experiments are chosen to minimise the impact of any inaccuracies this may cause (Section 4.3). It is expected to be a non-uniform cache architecture (NUCA), distributed across the chip.

Loki does not currently support hardware cache coherence; when coherence is required, it can be implemented in software using low-cost core-to-core message passing.

3.5 Programming Loki

Having a large number of cores and a high potential for computational throughput is not enough. It must also be feasible to effectively make use of the available resources. In Loki's case, this involves exposing functionality to the compiler and programmer.

3.5.1 Predicated execution

The Loki architcture supports predicated execution of all instructions to make simple control flow more efficient, and allow SIMD execution in the presence of slightly-divergent execution paths.

Many instructions have a `.p` variant, which stores a one-bit value in the core's predicate register. This value may be the least significant bit of the result of bitwise operations, the result

Encoding	Meaning
00	Execute if predicate = 1
01	Execute if predicate = 0
10	Always execute
11	End of packet (always execute)

Table 3.2: Predicate encodings.

itself in the case of Boolean operations, or the carry/borrow flag for arithmetic operations (see Appendix A for full details).

All instructions may be executed conditionally, based on the value of the predicate register. Two bits are reserved in the instruction encoding to determine when an instruction should execute. An `ifp?` prefix causes an instruction to execute only if the predicate is true, and `if!p?` instructions only execute when the predicate is false. The remaining two options are *always execute* (default), and *end of packet* (denoted with an `.eop` marker in assembly). This information is summarised in Table 3.2.

3.5.2 Remote execution

It is possible for one core to send instructions to another core or group of cores. Following a *rmtexecute* instruction, all consecutive instructions marked with an `ifp?` predicate will be sent to the specified destination, and will not be executed locally. This can be used to execute a small amount of code in parallel with the local instructions, or to have the remote core(s) fetch their own code sequence(s) and begin execution of an entire program. In some cases it will be cheaper to send instructions to a remote core which has cheaper access to the data than it would be to bring the data towards the local core.

All instructions sent remotely have the `ifp?` predicate replaced with a `.eop` end of packet marker, so that the remote core switches to an alternate source of instructions as soon as they begin to arrive. For this reason, the *instruction fetch* must be the final instruction sent to the remote core, otherwise the incoming individual instructions would be indistinguishable from the packet being fetched.

3.5.3 Parallelism

The Loki architecture was designed for cores to work together to execute an application. Although it is possible for a single core to execute any program by itself, the common case is expected to involve a large number of cores working together in a way which is tailored to each individual program. I call this co-operation structure an *overlay* or *virtual processor* as a reference to similar concepts on FPGAs and CGRAs, as it is possible to hide the details of the underlying simple cores and only ever consider the mechanism built on top of them.

With all of the features described above, Loki is a very flexible platform. It is possible to rapidly set up custom communication networks which match the flow of data within a program, and "reconfigure" the system in software as the program moves through its different phases. It has been shown that the parallelism available in different phases of a single program can vary massively [101]. The communication networks allow many forms of parallelism using the same underlying hardware: Loki can exploit instruction-level parallelism, data-level parallelism and thread-level parallelism simultaneously and often in multiple different ways, allowing the

```
pipeline {
  int_source_init(0) @ P0; // initialise data source
  while(1) {
    int n = int_source() @ P0;
    fifo_put(f1, n);
    n = fifo_get(f1);
    int_printer(n) @ P1;
  }
}
```

Figure 3.9: Loki-C code sample of pipeline parallelism. The calls to `fifo_put` and `fifo_get` mark the boundaries between pipeline stages and show how data is passed. The `@` notation places computation or data on a particular processor (which may be built out of multiple cores).

programmer or compiler to choose a mix of parallelism which best suits the current application. Some of these *execution patterns* are explored in Chapter 6.

I believe that software specialisation is a better solution than heterogeneous hardware: the architecture is simpler (which brings many further benefits), and a wider range of applications can be accelerated. Newer versions of a program can be accelerated by simply compiling them for the Loki architecture.

3.5.4 Loki-C

Although programming languages are beyond the scope of this work, a brief description of *Loki-C* is given here for completeness. Loki-C is a dialect of C which includes additional annotations which allow the programmer to express parallelism. It is strongly influenced by SoC-C (System on Chip C) [108], and includes constructs for pipeline parallelism, fork-join parallelism and data-parallel loops. There is also the ability to place computation on a particular core or virtual processor, and place data in a particular memory or virtual memory. Sample code demonstrating a software pipeline is given in Figure 3.9.

All benchmarks executed in this thesis, however, make use of standard C code, with the addition of inline assembly code to access features of which the compiler is not yet aware.

3.6 Limitations

As described, the Loki architecture has a number of limitations. Some of these are due to efforts to promote simplicity, and others allow the scope of the project to be constrained to make it more manageable. Some of these limitations are discussed below:

- No branch prediction hardware. Although this is not much of a problem for a single core because of the low penalties of cache misses and the ability to fetch instruction packets in advance, it may be difficult to exploit large amounts of instruction-level parallelism across multiple cores.
- No hardware cache-coherence. It is anticipated that it would be possible to use shared memory and message passing in most cases. There exist a number of schemes which greatly reduce the amount of required coherence information by using additional information, such as whether data is shared or private [66, 112].

- It would be difficult to use a traditional operating system on Loki because of its lack of software interrupts and virtual memory. These are subjects of future work. Also in this category are issues of protection: it is possible for a rogue program to send messages to arbitrary destinations, congesting the network and possibly providing incorrect values to running programs. It has been suggested that special privileges should be required to update a channel map table to avoid this situation.

For many of these limitations, it is possible to emulate the missing functionality in software (perhaps using additional cores), but the overheads of doing so are likely to be prohibitively high.

3.7 Summary

Loki is a flexible, low-power, simple architecture, designed to allow cores and memory banks to cooperate to execute applications efficiently. The network is central to the design, and is connected directly to the pipeline and exposed to the compiler and programmer.

Functionality is *accessible*, *bypassable* and *composable* wherever possible to reduce energy and increase cooperation. It is possible to build *overlays* on top of groups of components which match the natural structure of a program or phase of a program.

Chapter 4 explores the energy and area characteristics of each component of the Loki architecture in greater detail, and Chapter 5 uses these data to choose the best implementation.

CHAPTER 4

EVALUATION METHODOLOGY

In this chapter I describe the techniques used to collect all results reported in this thesis and evaluate the base design. The chapter builds up to a collection of energy and area models for all of the major components of the Loki architecture; Chapter 5 then aims to use this information to determine which implementation of each component is best.

4.1 Performance modelling

I make use of a custom SystemC [98] simulator to model the architecture and execute programs. SystemC is a library for C++ which provides an event-driven simulation kernel. The simulator is fast enough to run real programs whilst collecting important data: on the order of 100000 instructions can be executed per second, and this figure is largely independent of the number of cores being simulated. Data collected include event counts used for estimating energy (such as the number of times each register is read, or the number of bits which toggle on a given wire), and analytical data which can be used to pinpoint bottlenecks (such as the number of cycles each core is stalled, and the reasons for being stalled).

I examined a number of other simulators before settling on SystemC, including OMNeT [97], gem5 [20] and HASE [129]. I wanted more flexibility than many of them offered, as the Loki architecture is very different to the designs which are typically simulated, and I wanted lower-level information than most of them provided, in order to estimate energy consumption.

Simulation is cycle-accurate apart from the modelling of system calls, which complete instantaneously. For this reason, some benchmarks were lightly patched to remove system calls from inner loops, to reduce the impact on the results collected. These modifications are described in Section 4.3.

The level 2 cache is not fully modelled: it has a latency of ten cycles (beyond the L1), consumes no energy, and is large enough to hold all data required to execute a benchmark. The chosen compute-intensive embedded benchmark suite (Section 4.3) meant that this was not a major restriction.

4.2 Compiler

It was decided early on that relying on hand-coded benchmarks would not be acceptable, as this would make it very difficult to make fair comparisons with other architectures. It would be difficult to ensure that we weren't spending a disproportionate amount of time optimising for one architecture, which would skew any results.

For this reason, a compiler is used. This allows standard programs to be executed and more-direct comparisons to be made. The programs can also be larger and more interesting, allowing more to be learned from their execution behaviour.

The LLVM-based compiler has front-ends for a number of different programming languages, including C and Loki-C (Section 3.5.4), and outputs assembly code or binary executables for the Loki architecture.

Unfortunately, the compiler as used was not able to perform any architecture-specific optimisations. This meant that while the emitted code was functional, it was not of a particularly high quality. Details of how the impact of this was reduced are given in Section 4.3.1.

4.3 Benchmarks

I examined a number of benchmark suites before choosing MiBench [49]. MiBench is free (in both senses of the word) and has a relatively low memory footprint. The low memory footprint is important in an evaluation which largely excludes the memory system – having a large portion of the working set cached will make comparisons fairer. MiBench's programs are also relatively small and simple. This could be seen as a disadvantage because it means that some types of program may not be seen, but it was advantageous in this case because it reduced the amount of time required to become familiar with the benchmarks, and it increased the proportion of programs which were able to compile using the Loki toolchain.

Only integer benchmarks were used, since Loki does not yet have hardware floating point support, and only those benchmarks which compile (some require libraries which are not yet supported on Loki). This selection process left ten benchmarks which covered all six of the MiBench categories: automotive, consumer, network, office, security and telecom. The benchmark selection is compared against that of other projects in Table 4.1.

All benchmarks are compiled using the Loki compiler, using the settings suggested by the MiBench Makefiles. All benchmarks are executed using the "small" inputs in order to give reasonable simulation times. The newlib library for embedded systems provides standard library functionality [60].

Since the compiler is not yet able to perform architecture-specific optimisations, a second version of each benchmark is generated where the most frequently executed regions of code have had their assembly code hand modified. The modifications are expected to be within reach of a standard optimising compiler, and are described in Section 4.3.1. This allows a more-direct comparison between Loki and other architectures, for which the compilers are able to perform more aggressive optimisations. No modifications were made to library functions, so some of the benchmarks do not see much benefit.

The following sections give a brief overview of each benchmark.

adpcm

Adaptive differential pulse-code modulation is used in telephony as a means of signal compression. Both the encoder (*adpcmc*) and the decoder (*adpcmd*) are used.

The encoder receives a sequence of 16-bit samples of a sound wave as input, and converts each to a 4-bit output. A predicted value for each sample is generated using the preceeding sample, and the difference between the predicted and actual values is encoded using a look-up table. The decoder performs the reverse transformation.

Project	Benchmarks	Source
AsAP [128]	3: JPEG encode, MPEG encode, WLAN receiver	Self-written
Conservation cores [130]	5: mcf, JPEG encode, JPEG decode, bzip2, vpr	Various, including SPEC
Elm [14]	10 kernels, 6 benchmarks	Self-written
Loki	10: ADPCM encode, ADPCM decode, bitcount, CRC, dijkstra, JPEG encode, JPEG decode, qsort, SHA, stringsearch	MiBench
PPA [101]	3: MPEG audio decode, MPEG video decode, 3D rendering	Unknown
Raw [125]	14: adpcm, aes, btrix, cholesky, fpppp, jacobi, life, moldyn, mxm, sha, swim, tomcatv, unstruct, vpenta	Self-written, various SPECs and Mediabench
Smart Memories [82]	4: FFT, FIR, convolution, DCT	Self-written

Table 4.1: Comparison of benchmark suites

bitcount

The *bitcount* benchmark consists of several different techniques for counting the number of bits set in a large number of integers. For some experiments, the different counting methods are examined separately.

The program consists of an outer loop which traverses through the different bit counting functions and an inner loop which goes through all of the integers whose bits are to be counted. Some of the bit counting functions contain a further loop to check each bit or group of bits and update the total count.

There is a huge amount of parallelism in this program – virtually all of the loop iterations are independent. The loops are also very tight.

crc

The cyclic redundancy check is a hash function used to detect accidental changes in data. The program consists of a tight loop where each character of an input file in turn is hashed with the current checksum to produce a new checksum.

The program has been patched to remove an expensive library call from the inner loop: data is read from the input file in blocks of 1024 bytes, rather than one character at a time.

dijkstra

An implementation of Dijkstra's algorithm used to find shortest paths in a graph of connected nodes. The algorithm works by growing a tree of shortest paths out from a source node until the destination is reached.

The program has been patched to remove expensive memory allocation operations from the inner loops and instead statically allocate a fixed amount of space on the stack.

jpeg

JPEG is a common image compression technique. It is a relatively large and complex program with several stages of execution. Some stages are very sequential and control-intensive, whereas others consist of highly parallel loops.

We make use of both the compression (*jpegc*) and decompression (*jpegd*) programs.

qsort

An implementation of the quicksort algorithm, used to sort a list of 10000 strings. The program makes use of the standard library implementation.

Quicksort works by repeatedly splitting the list in two based on whether the strings should come before or after a chosen pivot element. Splitting stops when a list length of zero or one is reached, as the list is trivially sorted.

sha

The secure hash algorithm produces a 160-bit message digest from a file, used to detect data corruption or file tampering. It is used in the secure exchange of cryptographic keys.

Contents of a file are processed in 512-bit chunks, where each chunk is repeatedly permuted and transformed before being added to the digest.

stringsearch

An implementation of Pratt-Boyer-Moore string search, used to find a set of target strings in a set of text sequences.

The algorithm starts by initialising a table of values, where each value tells how far it is safe to skip through the text sequence if a particular character is seen. For example, if the character "x" is observed, but "x" is not in the target string, then it is safe to skip ahead by the length of the target string, as there can be no overlap with the target string and the current position. The algorithm then scans through the text sequence until the target string is found, or the end of the sequence is reached.

4.3.1 Optimisations

Hand-optimised versions of each benchmark were produced to make up for the lack of optimisations performed by the compiler. The modifications made were simple, applied only to the main loop bodies, and expected to be within reach of a more-mature optimising compiler. In fact, a compiler is expected to be able to do even better, as it would be able to perform a better register allocation after applying the optimisations. Register allocation was deemed too large a job to be done by hand. A brief description of each optimisation applied is given here.

- Improved support for Boolean values using Loki's comparison operations.
- Make use of the hard-wired register 0, rather than temporarily storing 0 to another register.
- Remove unnecessary no-ops.
- When a *fetch* instruction targets another *fetch* instruction, fetch the final target directly.
- Make better use of network channels: try to read directly from the channel, rather than first copying the data into a register, and don't use a separate instruction to send data.
- Fill load and branch delay slots with independent instructions, where possible.

Figure 4.1 shows the difference that these optimisations make. Averaged across all benchmarks, dynamic instruction count is reduced by 19% and execution time drops by 21%. The execution time generally decreased by more than the instruction count because filling the fetch and branch delay slots reduces the time spent idle: average instructions per cycle improved from 0.70 to 0.72.

Some benchmarks, such as *quicksort*, are not affected much by the optimisations because most of the execution time is spent in library code, which was not subject to optimisation. Others, such as *bitcount* and *crc*, improved by much more – up to a 1.8× speedup. This is because these benchmarks spend a lot of their time in very tight loops, so any reductions in instruction count have a large relative effect.

Table 4.2 presents the raw instruction count and execution time for each of the optimised benchmarks.

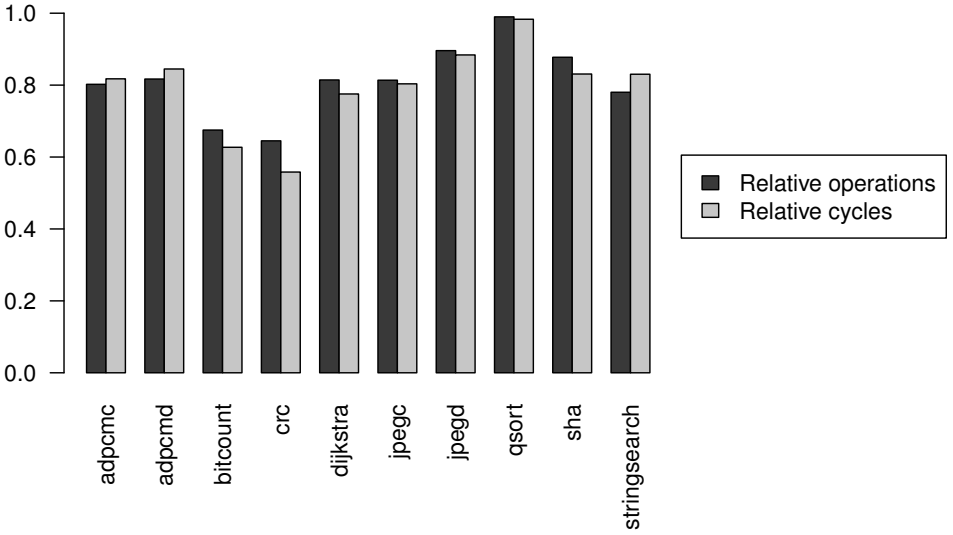


Figure 4.1: Effects of hand-optimisations on the MiBench suite.

Benchmark	Instruction count	Execution time/cycles
adpcmc	56,959,692	78,732,189
adpcmd	50,411,841	77,568,713
bitcount	58,892,272	75,602,682
crc	22,388,845	25,997,605
dijkstra	50,767,703	64,411,581
jpegc	56,029,375	83,614,501
jpegd	12,336,715	18,221,286
qsort	41,409,545	70,450,895
sha	19,930,572	24,057,591
stringsearch	485,258	796,533

Table 4.2: Benchmark execution characteristics, after optimisations have been applied.

4.4 Energy modelling

Energy modelling has become an important part of the modern chip design process due to the strict power limitations imposed by today's operating environments. Computer architects need to know early in the design process whether their architecture will meet the constraints so that they can make any necessary modifications while the design is still relatively flexible.

There exist many tools for high-level estimation of power consumption for computer architectures. McPAT [79] and Wattch [27] are general-purpose simulators, while Orion [61] focuses on the on-chip network and CACTI [92] models memory systems. Once again, however, Loki's design was found to be too unusual for any of the popular tools to be worthwhile: the scale of the modifications required would amount to a complete reimplementation of the Loki architecture. In addition, many of the tools target designs similar to modern commercial processors, and their models break down for structures as small as those used by Loki. I instead sought a bespoke energy-modelling framework which did not require reimplementing the entire architecture in a low-level HDL, and which had support for parameterised models.

In order to generate energy models for the Loki architecture, all main components were implemented separately in SystemVerilog and a standard-cell synthesis toolflow was used. Memories (including registers, buffers, etc.) are the main consumers of energy, and these are all implemented fully. Commercial memory and register file compilers and a commercial 40nm low-V_t design process were used. The width of each component of cores and memory banks was constrained to be 125µm, so that eight of them could sit side-by-side in a 1mm square tile. (1mm was chosen as a sensible starting point using initial estimates of the sizes of cores and memory banks.) All input ports to each module were forced to be on the top edge of the synthesised block, and all output ports were on the bottom edge.

Implementing each component individually speeds the design process: when designing a complete system, large amounts of time can be spent getting the different components to work together properly. It also allows us to see a breakdown of where the energy is being used. This allows optimisations to be targeted at the components which consume the most energy and comparisons can be made between different implementations of the same component more easily. It is also hoped that the modules are generic enough that the results can be useful to others.

This approach does, however, result in inaccuracies. The model for each individual component is likely to overestimate energy consumption; when synthesising a whole system, the tools are able to perform cross-boundary optimisations, which potentially reduce communication distances and relax critical paths. Conversely, since only the main components are modelled, there will be peripheral logic and wiring which is not taken into account, resulting in underestimation of total energy consumption. I believe that the impact of these unmodelled components is negligible – logic consumes much less energy than storage structures (see the decoder's model in Section 4.5 for an example), and none of the unmodelled wires are believed to be of significant length. Indeed, it has been shown that module-level models can be combined in this way to approximate total system energy consumption [115].

In order to make satisfactory progress, little time was spent on improving component designs: implementations are not naive, but do not involve obscure optimisations. Throughout the evaluation of the architecture, such optimisations are explored where it is evident that a particular component consumes resources disproportionately. All remaining optimisations are considered orthogonal to this work and can be applied in combination with any of the techniques presented.

All results are obtained for a commercial 40nm low-V_t design process. Only low-V_t cells are selected, and so leakage energy is insignificant and not reported. Timing is closed using a multi-corner PVT analysis (taking into account a range of voltages, ambient temperatures, etc.) where the worst case is usually the corner with the lowest voltage (0.99V) and the lowest temperature ($-40°$C). Energy results are then reported for the typical case (1.1V, 25°C), with the tools having been instructed to target the lowest possible dynamic energy.

For each implementation of each component, the following steps are taken:

1. Perform a thorough functional verification with Synopsys VCS.
2. Synthesise using Synopsys Design Compiler.
3. Place and route using Synopsys IC Compiler.
4. Extract parasitics using Synopsys StarRC.
5. Analyse timing (taking parasitics into account) using Synopsys PrimeTime.
6. Simulate using VCS, logging important events.
7. Estimate energy consumed each cycle using PrimeTime.
8. Combine the event log and energy consumption data to form an energy model using R.

A 435MHz clock rate is targeted due to simultaneous constraints from the instruction packet cache, register file, and memory bank. The design is conservatively margined at the worst case corner, taking into account foundry recommendations for on-chip variation (OCV) and clock jitter: the longest logic paths have their timing derated by 8%, the shortest paths have their timing derated by -10%, and clock uncertainty is set to 35ps. Loki's clock period is roughly 42 FO4 delays, which is within the typical range of 40-60 used by modern system-on-chip designs. I anticipate that the common target of 500MHz could be reached with the low-level tuning which is typically applied to production designs, and further improvements could be made with the standard practice (for mobile devices) of reducing the operational temperature range and other margins.

The timing of components is constrained such that all outputs are produced within the 2.3ns clock period. For some components, the delay is constrained to half a cycle so that two components can be activated in series in a single cycle. An example of this is the L0 cache tag check in the first half of a clock cycle, followed by the data array access in the second half. Small timing violations (less than 0.05ns) are permitted as these are likely to be covered by the generous margins applied and could probably be eliminated by small tweaks to the design or toolflow. For some components, such as network buffers, the timing constraint is set to 0ns to force the tools to produce the fastest circuit possible. In general, a faster circuit will require more redundancy and larger transistors, and so the circuit becomes larger and more energy-hungry. Since the very fastest circuit is rarely needed, this can result in a slight overestimation of area and energy consumption.

In stage 6, a wide range of inputs are supplied (usually randomly, but data collected from an execution trace can also be used), and a summary of all high-level events which occur each cycle is printed. For example, for a register file, the following information is collected:

- Whether there was an operation on each of the read and write ports

- The number of bits toggled on each port
- The number of bits which are high on each port
- Whether each of the reads resulted in a data bypass

In stage 8, an event summary (stage 6) is matched up with an energy figure (stage 7) for each clock cycle. A multiple regression analysis is performed on this data, assuming that all events are correlated with energy. If the analysis is unable to find a correlation with a particular event, that event is removed and the analysis is repeated. A low intercept is targeted, as this means the events of each clock cycle make up the majority of energy consumed; the low-leakage process used means that static power is negligible, so the intercept should represent only the internal clocking network of each component. If the intercept is sufficiently large, the design and the energy consumption are inspected, in an attempt to find further events which may contribute to energy.

The residual errors of the model should be roughly normal – the Shapiro-Wilk normality test can be used to test for this [117]. A normal distribution of errors makes it easier to estimate the cumulative error for a large number of events. In practice, I found it difficult to achieve distributions normal enough to pass the test, and settled for distributions which visually looked close to normal (Figure 4.2). An obviously skewed or multi-modal distribution showed that there was relevant information not being used by the model, so another iteration of model generation was required. For the purposes of error analysis, the distribution of errors was approximated by a normal distribution with the same mean and variance as the true distribution.

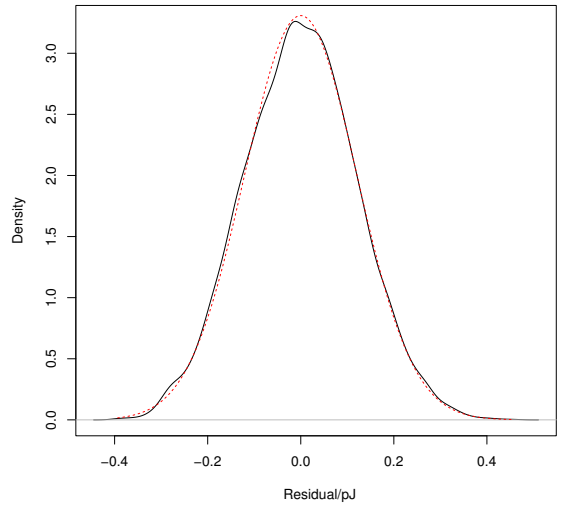
For some models, such as the register file, *spacer cycles* are required to generate an accurate model. This is because when writing to a register, the value is only latched on the following clock edge, and so appears in the following cycle's energy data. For these models, I follow each cycle of activity with an empty cycle, sum the energy consumed by both cycles, and halve the intercept computed by the regression analysis.

Since the events of interest are independent of each other and of when they occurred, it is possible for the simulator to maintain a counter for each event and only output a summary of all events when simulation finishes. This allows execution to be instrumented with only a small impact on simulation time. There is no need to apply the energy models during simulation – indeed it is preferable to have a trace which can be analysed offline, as this allows different models to be applied to the same trace without needing to repeat the simulation.

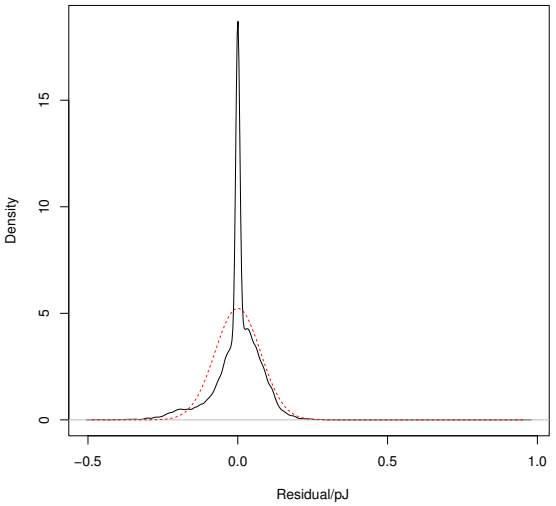
4.5 Models

In this section, a model is presented for each of the major components on the chip. Most implementations are expected to be generic enough to be applicable to other architectures as well. Exceptions are modules with Loki-specific behaviour, such as the instruction decoder. Each model consists of the following information:

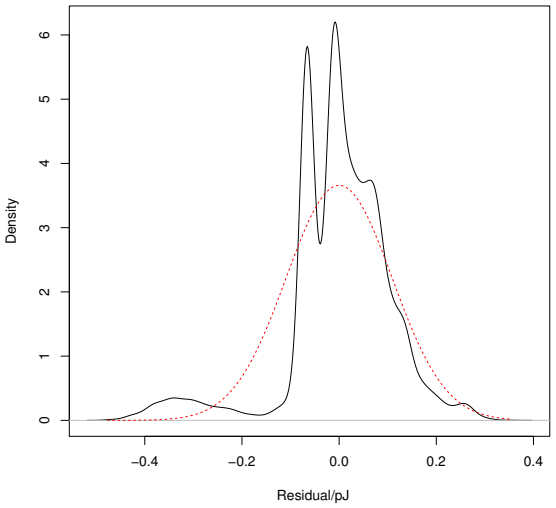
- a summary of the information required to compute energy consumption;
- an energy equation, where each event is associated with an energy cost (in picojoules);
- an estimate of the average error of the energy model in a single cycle – the residual standard error (RSE), which is the standard deviation of the distribution of errors;



(a) A good error distribution generated for a 32-bit bus.



(b) A mediocre error distribution generated for the register file. Despite the slight negative skew, the large peak around 0 gave me confidence that the model was usually very accurate.



(c) A poor model which needs refinement. The extra peaks indicate that important information is not being included in the model.

Figure 4.2: Error distribution examples. Each true distribution (solid) is compared with a normal distribution with the same mean and variance (dashed).

- the area consumed by the placed-and-routed design.

All models are produced for a 40nm low-leakage process, and target a 435MHz clock frequency. Perfect clock gating is assumed – components are modelled as consuming zero energy in cycles during which they are not used. The combination of the low-leakage process and clock gating mean that the energy of idle components is a fraction of one percent of the total energy consumed, and is ignored in the models.

There is a trade-off between the complexity of a model and its accuracy. The more complex a model is, the harder it is to compare it to another model, and the higher the overhead of collecting the information during simulation. However, complex models tend to be more accurate and provide more information about exactly where and when energy is being consumed. Increasingly complex models have diminishing returns; doubling the number of variables often only improves the error of the model slightly.

In this section, models trend toward the simple end of the scale, and have only a few variables each. The law of large numbers states that as long as the average error of each model is zero, then the expected error of many applications of that model will tend towards zero, so a slight increase in the error of each model is not an issue.

In order to apply an energy model to an execution trace, each event count should be multiplied by the energy cost provided by the model. The event count could be a sum of Hamming distances (or similar) or a sum of Boolean values (with 1 representing "true"). The residual standard error applies to the total energy consumption of the events which take place in one clock cycle.

There are some parts of the design which are not modelled (e.g. the predicate register), resulting in an underestimation of energy and area, but these parts are expected to provide a negligible contribution. All components containing a non-trivial amount of memory, complex logic, or long wires are modelled. The models are described individually in the following sections, with a summary of how they combine to form entire cores and tiles at the end.

4.5.1 ALU

The ALU receives 32-bit operands and produces 32-bit results for all operations. A list of supported operations is given in Table 4.3.

The model has a relatively large standard error of around 15%. This is because the ALU is a large mass of irregular logic which is difficult to simplify in a model. Attempts were made to reduce the error by collecting information for each operation or class of operations separately, but this only had a small effect and was not deemed worthwhile.

Example

In order to demonstrate how these models can be used to estimate energy consumption, a simple example is given here.

Suppose that in a particular clock cycle, the ALU performs an *add* operation. 8 bits change in the first operand from the value used by the previous operation, 13 bits change in the second operand, and 10 bits change in the output. The previous operation was not an addition. Energy can then be computed as follows:

Class	Mnemonic	Description
Comparison	seteq	Set if equal
	setne	Set if not equal
	setlt_s	Set if less than (signed)
	setlt	Set if less than (unsigned)
	setgte_s	Set if greater than or equal (signed)
	setgte	Set if greater than or equal (unsigned)
Shift	sll	Shift left logical
	srl	Shift right logical
	sra	Shift right arithmetic
Arithmetic	add	Addition
	sub	Subtraction
Bitwise logic	nor	Negated OR
	and	AND
	or	OR
	xor	Exclusive OR
	nand	Negated AND
	clr	Clear; AND with negated second operand
	orc	OR complement; OR with negated second operand
Miscellaneous	clz	Count leading zeroes
	lui	Load upper immediate

Table 4.3: Supported ALU operations

Variable	Description
op	Operation occurred this cycle
hd_in1	Hamming distance of first operand
hd_in2	Hamming distance of second operand
hd_out	Hamming distance of output
same_op	Operation was the same as the previous one
high_energy	Operation was one of {add, setgte}

op	hd_in1	hd_in2	hd_out	same_op	high_energy	RSE	Area/μm^2
0.96	0.028	0.048	0.023	−0.39	0.34	0.38	2862

Table 4.4: ALU model

$$
\begin{aligned}
energy/pJ &= 1 \times 0.96 + 8 \times 0.028 + 13 \times 0.048 \\
&\quad + 10 \times 0.023 + 0 \times -0.39 + 1 \times 0.34 \pm 0.38 \\
&= 2.38 \pm 0.38
\end{aligned}
$$

4.5.2 Arbiter

Arbiters are used throughout the network to decide which data should proceed in the case of resource contention. An arbiter is required at the input of each memory, and two arbiters are required at the input of each core, since each core can receive two words per cycle from the network.

A simple 8-input matrix arbiter was chosen for its speed and simplicity. There are many other types of arbiter to choose from, but none are expected to consume a large portion of the total energy or area, so it is believed that the matrix implementation is representative.

Variable	Description
activity	Any request or grant line is active

constant	activity	RSE	Area/μm^2
0.06	0.39	0.06	420

Table 4.5: Arbiter model

The latency of the combinational logic is 290ps, even with much more time available. It is expected that this is the most energy-efficient implementation the tools could generate, and that it would be possible to generate a faster but higher-energy arbiter if necessary.

4.5.3 Clock

The clock network can consume a large fraction of the total energy used by a microprocessor, so generating a model for it is important [69]. Each of the models for individual components includes the local clock network, so only the higher levels of the clock tree need to be modelled here.

Orion 2.0 [61] was used to model the high-level clock network as an H-tree with optimal buffering. The network spans a single tile of the Loki architecture, and the result was validated against a 1-bit bus of comparable length, whose model is presented in Table 4.12.

constant	RSE	Area/μm^2
1.37	-	-

Table 4.6: Clock model

Since the clock energy is a constant for each tile, it is biased towards the use of more cores, where the clock's energy will be a smaller fraction of the total. For this reason, the clock energy is omitted from total energy when comparing between two different implementations of the same benchmark running on Loki.

4.5.4 Crossbar

The crossbar carries 32-bit data and excludes all buffering and arbitration. Since the crossbar spans the whole tile, the placement of its pins can have a significant impact on the length of wires generated. The synthesis tools aim to reduce latency and energy consumption, so will want to place the pins as close together as possible, but this is not useful for connecting components which are physically separated. For this reason, the pins are manually placed, with each 32-bit word clustered together in the centre of each eighth of a millimetre.

Table 4.7 presents energy models for crossbars with different numbers of input and output ports. The area of each crossbar is manually specified to increase wire spacing and reduce cross-coupling capacitances, which results in better energy efficiency.

Variable	Description
total_dist	Length of wire toggled: Hamming distance × length in mm
hd_in	Total Hamming distance at all inputs
hd_out	Total Hamming distance at all outputs

Inputs	Outputs	total_dist	hd_in	hd_out	RSE	Area/μm^2	Latency/cycles
4	4	0.061	-0.012	0.017	0.13	500×30	0.5
4	8	0.0006	0.042	0.007	0.07	750×30	0.5
8	4	0.034	-0.013	0.030	0.36	750×30	0.5
8	8	0.057	0.014	0.033	0.56	1000×30	0.5
8	16	0.0043	0.137	0.015	0.65	1500×60	1
16	8	0.0041	0.038	0.055	0.69	1500×60	1
16	16	0.0041	0.046	0.019	0.41	2000×60	1

Table 4.7: Crossbar model

4.5.5 FIFO buffer

FIFO buffers are used in two main situations in the Loki architecture: in the network to tolerate bursty traffic, and in the core as a place to hold instructions and data which do not need to be cached. The network buffers must be fast enough to allow data to travel from one buffer, along a length of interconnect, and into another buffer, all in half a clock cycle. In contrast, the instruction buffer does not share the critical path with anything significant, so can be slower and more energy-efficient. Separate models for each type of buffer are presented below.

Table 4.8 demonstrates the effect that timing constraints can have on a design. The fast 16-entry FIFO is much more expensive to read from than the slow 16-entry FIFO, and its area is 59% higher. The fast FIFO is cheaper to write to than it is to read from, whereas the slow FIFO is cheaper to read from. The cost of writing to either implementation is similar. It is expected that this is because the read logic is on the critical path, so is modified by the synthesis tools in an attempt to meet the stricter timing constraints.

Curiously, the models suggest that a buffer with four entries consumes less energy than a buffer with two. This result was consistent across multiple syntheses, but its cause is unknown and appears to be anomalous.

Variable	Description
entries	Number of 32-bit spaces in the buffer
push	Data was written this cycle
pop	Data was read this cycle

entries	constant	push	pop	RSE	Area/μm^2
2	0.10	0.66	0.39	0.05	565
4	0.09	0.61	0.33	0.05	920
8	0.24	0.48	0.49	0.07	1946
16	0.25	0.58	1.01	0.12	4539
8	0.10	0.58	0.25	0.04	1454
16	0.11	0.68	0.31	0.06	2861

Table 4.8: FIFO models – the first four lines are for the fast network buffers, and the final two lines are for the efficient instruction buffer.

4.5.6 Instruction decoder

The decoder is capable of extracting all fields from all instruction encodings. Random valid instructions were used to generate the model.

constant	RSE	Area/μm^2
0.52	0.12	555

Table 4.9: Decoder model

The energy consumed by the instruction decoder is largely independent of the input instruction, so a simple constant value could be used. The large relative error was considered acceptable because the decoder uses such a small fraction of total energy.

4.5.7 Instruction packet cache

The instruction packet cache is modelled in two parts: the tags, to be checked in the first half of the clock cycle, and the data array, to be read (if necessary) in the second half.

Cache tags consume large areas (Table 4.10), but due to their highly parallel nature, even large numbers of tags can be accessed within half a clock cycle.

The data array was implemented using a commercial register file generator, with one read port and one write port (Table 4.11). The tool allowed multiple configurations for each size of array; these configurations were explored and only the best implementation is shown here for each size. The 256 entry cache did not meet the timing constraints, but is shown here for comparison.

There is a steep jump in area and energy between the 64 and 128 entry caches because a configuration parameter governing arrangement of multiplexers had to change in order to meet timing constraints. A similar change took place between the 128 and 256 entry caches.

Variable	Description
tags	Number of 30-bit tags
read	Tags were checked this cycle
write	Tag was written this cycle
hd	Hamming distance between consecutive tags being searched for

tags	constant	read	write	hd	RSE	Area/μm^2
1	0.00	0.02	0.58	0.006	0.04	283
2	0.01	0.02	0.59	0.014	0.04	587
4	0.05	0.03	0.57	0.027	0.04	1101
8	0.08	0.07	0.52	0.042	0.05	2056
16	0.12	0.15	0.50	0.079	0.07	4000
32	0.33	0.16	1.15	0.198	0.13	7970
64	0.54	0.35	1.94	0.426	0.27	16968

Table 4.10: Cache tag models

Variable	Description
entries	Number of 32-bit spaces in the cache
read	Data was read this cycle
write	Data was written this cycle

entries	constant	read	write	RSE	Area/μm^2
32	0.12	2.28	2.23	0.06	2583
64	0.12	2.40	2.35	0.06	3623
128	0.15	3.95	4.19	0.08	6120
256	0.14	5.33	5.90	0.12	9695

Table 4.11: Instruction packet cache models (excluding tags)

4.5.8 Interconnect

A 2mm long, 32 bit bus was modelled to explore the costs of communication in an architecture with such an emphasis on its network. Energy and delay were then normalised to a 1mm bus. This approach was used to avoid any irregularities in the model at shorter wire lengths, allowing a linear model to be generated which can scale to longer wires. The wire spacing was controlled by placing additional constraints on the width of the bus.

Variable	Description
target	Optimisation target of synthesis tools
hd	Hamming distance between successive values on the bus

target	Channel width/µm	hd	RSE	Delay/ps/mm	Wire density
speed	2.5	0.093	0.026	345	45%
speed	5	0.082	0.027	325	22%
speed	10	0.078	0.027	310	11%
energy	2.5	0.070	0.024	735	45%
energy	5	0.064	0.026	710	22%
energy	10	0.057	0.019	680	11%

Table 4.12: Interconnect model

Table 4.12 shows that optimising for speed rather than energy consumption approximately doubles transmission speed and increases energy consumption by around 30%.

The spacing between wires also has an effect: the closer the wires are together, the higher the cross-coupling capacitances, and the higher the energy and delay. The densest buses shown here consume around 20% more energy and are around 10% slower than the sparsest ones, but occupy one quarter of the area. The minimum possible area for a bus is determined by the minimum wire pitch of the technology and the number of metal layers available for sending data in a particular direction:

$$\begin{aligned} area &= \frac{bus\ width \times minimum\ wire\ pitch \times bus\ length}{metal\ layers} \\ &= \frac{32 \times 0.14\mu m \times 1000\mu m}{4} \\ &= 1120\mu m^2 \end{aligned}$$

Double-spacing (50% wire density) is often used as a compromise between area and energy consumption, and these figures show that further savings can be made if extra time or area are available.

The figure for a fast, dense bus of approximately 0.1 pJ/bit/mm correlates well with other published results [45]. Since Loki's clock period is 2300ps, and tiles are approximately 1mm×1mm, it may be possible for data to travel across multiple tiles in a single clock cycle, even in the slowest case, reducing the diameter of the network.

4.5.9 L1 cache bank

Each L1 cache bank can hold 8kB of data. The cache is direct-mapped and has 32-byte cache lines. In order to improve access time, the data array is divided into four identical 2kB sub-banks, generated using a commercial memory compiler.

Variable	Description
w_{rd,wr}	Word (32-bit) read/write
hw_{rd,wr}	Halfword (16-bit) read/write
b_{rd,wr}	Byte (8-bit) read/write
ipk_rd	Read a single word from a streaming instruction packet
rep_line	Replace an 8-word cache line

w_rd	hw_rd	b_rd	ipk_rd	w_wr	hw_wr	b_wr	rep_line	RSE	Area/μm^2
9.01	9.27	9.45	6.88	12.00	19.28	19.00	60.16	0.3	52500

Table 4.13: L1 cache bank model

Table 4.13 shows that streaming words in an instruction packet is almost 25% cheaper than reading words individually. This is because the cache tags only need to be checked once for each cache line, rather than for each word, and because fewer address bits switch internally. This suggests that a mechanism for streaming data would also be useful; exploration of this is left for future work.

Sub-word accesses are more expensive than word accesses because of the required masking and shifting. Sub-word writes are particularly expensive because data must be read from the data array, updated, and written back. These expensive operations make up only 3.6% of all memory accesses across the benchmarks used, so are not considered for optimisation.

4.5.10 Multicast network

The multicast network is a crossbar with 8 inputs and 8 outputs, made up of buses described in Section 4.5.8. Pins were placed in the appropriate locations using the same technique as the Crossbar model (Section 4.5.4), and multiplexers were placed at all outputs to ensure that enough load was placed on the buses. Again, spacing of the wires had a significant impact on latency and energy consumption, so the smallest implementation where these effects were negligible was chosen, giving the results in Table 4.14.

Variable	Description
hd_in	Total Hamming distance at all inputs
hd_out	Total Hamming distance at all outputs
hd_select	Total Hamming distance of all multiplexer selections

constant	hd_in	hd_out	hd_select	RSE	Area/μm^2
0.020	0.045	0.020	0.205	1.08	30×1000

Table 4.14: Multicast network model

4.5.11 Multiplier

Loki's multiplier takes two 32-bit operands and is capable of computing either the upper or lower word of the result in two clock cycles. Table 4.15 presents its simple energy model.

constant	RSE	Area/μm^2
6.43	1.70	7429

Table 4.15: Multiplier model

4.5.12 Pipeline register

Pipeline registers may be small, but there are many of them, and they are active often. As a result, they can consume a significant portion of total energy of the chip (see Section 5.4). Table 4.16 shows that the area and energy consumption scales linearly as the pipeline register gets wider, as would be expected.

Variable	Description
width	Number of bits the register can store
write	Register was written this cycle
hd	Hamming distance between consecutive values in the register

width	constant	write	hd	RSE	Area/μm^2
32	0.01	0.22	0.004	0.002	121
48	0.02	0.32	0.003	0.002	179
64	0.04	0.41	0.003	0.003	238
80	0.04	0.51	0.003	0.003	295
96	0.06	0.60	0.003	0.003	354

Table 4.16: Pipeline register model

4.5.13 Register file

Although we have access to a commercial register file compiler, I settled on an implementation built from standard cells. This was for two reasons. First, the commercial compiler used is only capable of generating register files with a single read port, while Loki's register file requires two read ports. Second, the compiled modules trade energy for speed; they are much faster than standard cells, but also consume much more energy. Since Loki focuses more on energy efficiency than speed, and the slower standard cell implementation was still fast enough, it was chosen in preference of the compiled implementation. The energy model is presented in Table 4.17; a 16-entry register file is included to represent the channel map table.

4.5.14 Router

Each tile contains a router which links it with its four immediate neighbours. The router contains a buffer for each input, an arbiter for each output, and a 5×5 crossbar.

Variable	Description
entries	Number of 32-bit spaces in the register file
ports	Number of read and write ports available
wr	Data was written this cycle
rd$X$	Data was read from port X this cycle
*port*_hd	Hamming distance between consecutive words on *port*
*port*_oc	Number of "ones" in the data on *port*
by$X$	Input data was bypassed to read port X

entries	ports	constant	wr	wr_hd	wr_oc	rd1	rd1_oc	by1	rd2	rd2_oc	by2	RSE	Area/μm^2
16	1R1W	0.04	0.62	-	-	0.37	0.018	0.35	-	-	-	0.04	2915
32	1R1W	0.04	0.91	-	-0.009	0.42	0.027	0.41	-	-	-	0.08	5597
32	2R1W	0.11	0.78	0.012	-	0.39	0.026	0.51	0.39	0.026	0.42	0.11	6815

Table 4.17: Register file model

Since this work focuses on communication within a tile, the router is never used, so no energy model is provided. Only area is given in Table 4.18 as this is required to generate a tile floorplan.

Area/μm^2
100×100

Table 4.18: Router model

4.5.15 Scratchpad

Models for various sizes of scratchpad are presented in Table 4.19. They are implemented as memory banks with a single read/write port, which are allowed a single clock cycle to produce their results. The 32-entry scratchpad is built using standard cells to reduce energy costs, but larger ones proved prohibitively large and slow, so make use of a commercial memory compiler instead. Data arrays larger than 256 entries could not be accessed in a single clock cycle.

Variable	Description
entries	Number of 32-bit words in the data array
read	Data was read this cycle
write	Data was written this cycle

entries	constant	read	write	RSE	Area/μm^2
32	0.04	0.85	0.77	0.10	5597
64	0.12	2.68	2.46	0.11	2321
128	0.13	2.86	2.63	0.12	3271
256	0.15	3.50	3.29	0.14	5171

Table 4.19: Scratchpad model

Since the standard cell implementation is so much cheaper to access than the compiled versions, it may be worth producing a hybrid structure to take advantage of the low energy of the standard cells, and the high capacity of the compiled data arrays.

4.6 Summary

We are able to create energy models for each of the main components of a multi-core RISC processor with relatively low error. These models can be applied to a simple execution trace to get accurate figures for the whole design.

The law of large numbers states that the expected error of the sum of model applications will tend towards zero as the number of model applications increases. This assumes that the model was generated using the same data distribution as is seen during execution. In practice, this is not the case: significant correlation is typically observed in execution traces which is not seen in the random data used to generate the models. This correlation tends to reduce Hamming

distances, and therefore energy consumption, so the models are expected to represent a slightly pessimistic distribution of energy costs.

The following chapter experimentally determines suitable implementations for each component, and shows how all modules fit together to form the baseline Loki architecture.

CHAPTER 5

DESIGN SPACE EXPLORATION

In this chapter, I explore the various possible implementations of the major components of the Loki architecture, and attempt to develop a coherent overall design which maximises performance whilst remaining as energy-efficient as possible. I explore the design space of several components of the architecture, and also explore the performance and energy impact of using them in different ways.

This task is made challenging because Loki cores are designed to be used together and share resources, but all benchmarks in the MiBench suite target only a single core. An iterative approach of evaluation and refinement is used: in this chapter, I determine the best configuration for sequential workloads, and leave open the possibility of further tweaks when parallel workloads are explored in Chapter 6.

5.1 Instruction supply

Instruction supply makes up a large portion of energy spent in embedded processors – as high as 40-50% [33, 77]. This is because instruction fetching happens almost every cycle, rather than the 20-30% of the time for data loads and stores, with each fetch requiring access to a relatively large and expensive memory structure. Loki therefore aims to reduce these costs as much as possible. Traditionally, this is done by caching instructions locally. Storing instructions near to where they are needed reduces both the latency and energy costs of accessing them. Instructions typically exhibit high spatial and temporal locality, making caches particularly effective. On a modern chip, more than 50% of the die area can be made up of caches (for both instructions and data) [122]. However, while modern caches succeed in keeping a large proportion of instructions on chip, they are usually optimised for performance rather than power consumption. Their large sizes mean that they are expensive to access and data must travel a long way to get to and from the pipeline. Cache accesses typically cost at least an order of magnitude more than ALU operations [18, 33].

Loki provides a deeper memory hierarchy than usual with a very small and cheap L0 cache primarily aimed at reducing energy consumption. Extending the memory hierarchy is not a new idea; filter caches and loop caches have been around for decades to improve energy consumption of tight loops [70]. These small caches are optimised for energy consumption rather than performance, and can struggle when the working set is too large. For this reason, each Loki core also has an instruction buffer which allows the L0 cache to be bypassed in situations where it performs poorly and consumes more energy than it saves.

This section explores ways in which energy consumption (and to a lesser extent, performance) can be improved by choosing the right instruction storage structure for a particular situation.

5.1.1 Instruction packet cache

A previous study by Park et al. explored various implementations of level-0 instruction stores [102]. It was found that compiler management was the most effective, as cache tags could be eliminated and tight control over which instructions were in the store was possible. A direct-mapped cache was next-best, and finally, a fully-associative cache with FIFO replacement. FIFO replacement was a particular problem because if the body of a loop was even one instruction larger than the cache's capacity, the first instructions of the loop would be overwritten, and when they were refetched, they would overwrite further instructions which would be needed soon.

It may then come as a surprise that Loki makes use of a fully-associative L0 cache with a FIFO replacement policy. In part, this decision is forced by Loki's use of instruction packets; a FIFO replacement policy means that if the head of a packet is found in the cache, the rest of the packet will also be there, removing the need for extra tag checks and logic to implicitly fetch parts of packets which are no longer in the cache. It is also possible to prevent the current instruction packet from being overwritten by one in the process of being fetched by keeping track of how much space there is between the cache's read and write pointers and issuing the fetch only when there is enough space for the largest possible packet.

Loki's L0 cache has a number of features to address the concerns of Park et al., but it is also believed that complete compiler management is a sensible avenue for future investigation. First, tags only need to be checked once per instruction packet, rather than for every instruction or cache line, making their overhead negligible in many cases. Second, instructions are provided which allow partial compiler management of the cache (*fill*, *in-buffer jump*, etc., described in Section 3.2.1). Third, the instruction buffer allows the entire cache to be bypassed in situations in which it will not perform well (discussed below).

Removing the tags (or accessing them in parallel with the data array) would allow a whole cycle for data array accesses, which would allow the data array to be larger. I avoid this for a number of reasons:

- With no tags, explicit compiler management would be required, which is beyond the scope of this work.
- Accessing tags in parallel with the data array would not work with the fully-associative cache structure used.
- A larger cache is more expensive to access (the memory compiler used is not constrained by the clock period) and would raise the lower bound on energy-per-operation in many cases, as it is used almost every time an instruction is executed.
- There are ways of distributing a program over multiple cores to ease the pressure on a single core's cache. These techniques are explored in Chapter 6.

Figure 5.1 shows performance and energy consumption for caches with capacities between 32 and 256 words, and between 1 and 64 tags, for benchmarks which use the L0 cache exclusively. The better designs are towards the bottom-right of the graph. Increasing cache size

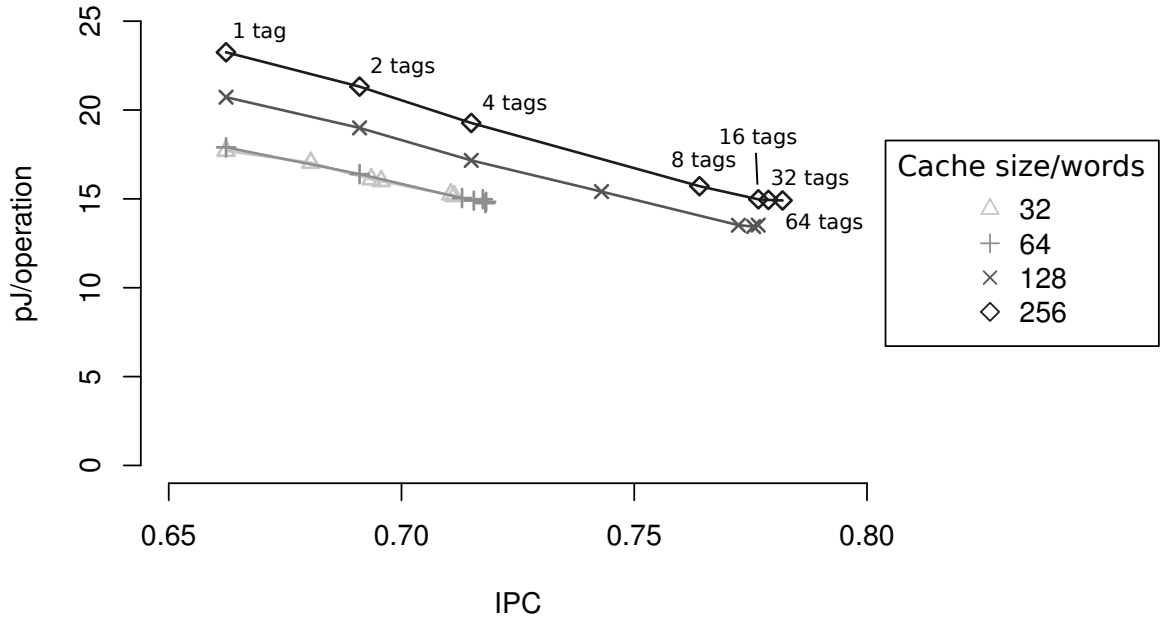


Figure 5.1: Normalised performance and energy consumption for different instruction packet cache sizes and numbers of tags. Tag counts start at 1 at the left of the graph and double with each step towards the right. The 256-word caches do not meet the timing constraints, and are presented for comparison only.

improves performance, as more of the working set can be stored locally. Energy consumption also improves up to 128 words (when there are enough tags), suggesting that this is a common working set size for these applications, and any further cache capacity does not help much. For each cache size, increasing the number of tags also improves performance and energy consumption because internal fragmentation is reduced, so the cache can be better utilised. Providing more than 16 tags gives only small benefits.

The data show that there are only a small number of Pareto optimal cache designs – designs for which there are no others which are better in both metrics. These are: 256 words with 64 tags, 128 words with 64 tags, and 128 words with 32 tags. When area is also considered, implementations with 16 tags are also attractive, since cache tags consume relatively large areas (Table 4.10), and only a small performance reduction is observed.

Table 5.1 shows cache hit rates for a range of configurations. This experiment was performed later than the one in Figure 5.1 and differs in that it covers 30 MiBench applications (rather than 10) and does not make use of hand optimisations to the benchmarks. In this case, the smaller 64-word caches are much more competitive, despite the lack of hand optimsations which tend to reduce the code size. This difference is down to the larger benchmark suite – the two *adpcm* benchmarks exhibit a step-change in performance when the cache size drops below the length of the main loop body, but almost all others degrade gracefully. Experiments in Chapter 6 demonstrate that applications can be distributed across multiple caches, allowing such cases to be eliminated.

A cache capable of holding 64 instructions and 16 tags was selected: even a slight reduction in either of these values results in at least a 3% drop in instruction hit rate, while doubling either of them provides only marginal improvements. With this configuration, an average tag look-up costs 0.66pJ, a cache read costs 2.5pJ and a write costs 2.4pJ.

Tag count	Cache size/words	Packet hit rate	Instruction hit rate
8	32	48.74%	41.82%
8	48	56.10%	50.57%
8	64	57.60%	53.51%
12	32	51.17%	43.66%
12	48	58.66%	52.50%
12	64	60.41%	55.79%
16	32	58.74%	49.30%
16	48	66.32%	58.21%
16	64	68.10%	61.52%
16	128	68.54%	62.43%
32	64	70.12%	62.94%
32	128	70.56%	63.85%
256	256	71.57%	67.98%

Table 5.1: Hit rates for instruction packets and individual instructions for a range of cache configurations. Data was kindly provided by Andreas Koltes and shows averages across thirty MiBench applications with no hand optimisations applied.[1]

5.1.2 Instruction buffer

The instruction buffer is a small instruction store used for instructions which will only be needed once. It was originally included as a way for cores to send commands to each other (and is still used for this purpose), but was also found to be a useful structure for other reasons.

When the active code section is small, the L0 cache reduces energy costs by reducing the number of L1 accesses required. However, when the L0 cache performs poorly (the active code section is large), the L0 itself adds a relatively large overhead – every instruction gets written to the L0 and read once before being overwritten. In these cases it is preferable to use the buffer as its access costs are much lower.

This section examines performance when the buffer is used exclusively; the next section explores switching between the cache and buffer to minimise energy. The largest available instruction buffer is selected as it is only marginally more expensive to access than one half its size, and the extra capacity will be useful in more situations. The buffer has 16 entries, with an average write costing around 0.75pJ and an average read costing around 0.35pJ.

Figure 5.2 shows the performance and energy consumption of benchmarks when they use the instruction buffer, relative to when using the selected IPK cache. As expected, the energy of some benchmarks (e.g. *adpcm*) reduces. *adpcm* consists mainly of a single loop which is too large to fit in the small L0 cache, and the buffer eliminates most of the overheads of supplying instructions within the pipeline. Writing to and subsequently reading from the buffer costs only 1.1pJ, compared with the L0 cache's 4.9pJ.

The energy of other benchmarks (e.g. *crc*) increases because they were already served well by the locality captured by the L0 cache. The additional costs of going back to the L1 cache outweigh the benefits of using a cheaper storage structure in the pipeline. For these benchmarks, the instruction buffer would not be used.

[1] More benchmarks were available for this experiment than in my own evaluations as more time had been spent on improving compiler compatibility, and because floating point applications were included.

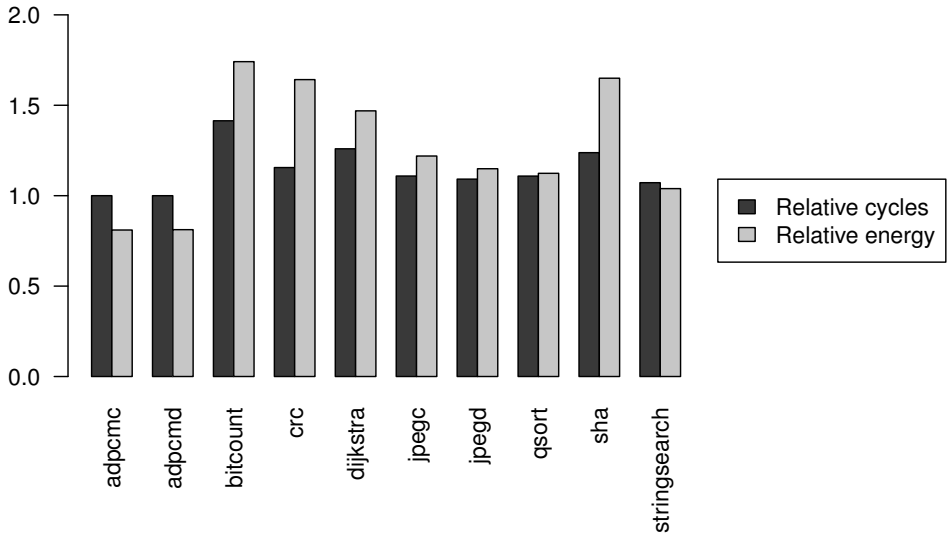


Figure 5.2: Behaviour of instruction buffer relative to L0 cache.

In more complex programs, the best structure to use will vary as execution moves through different stages. In general, small loops whose most-frequent execution paths fit in the L0 cache should use the cache, and all other code sections should use the buffer. The following section explores how this can be extended to make the cache profitable in more situations.

5.1.3 Cache pinning

Cache pinning (also known as cache locking) is a software-managed technique which aims to reduce cache conflicts in situations where the active code section is larger than the cache. This is done by the compiler *pinning* a block of instructions to a fixed position in the cache, and using any remaining space to cycle through all other required code. The pinned instructions never experience a conflict, and so the hit rate of the cache is improved. The technique is particularly useful for improving determinism in cache-limited systems such as domain-specific processors and embedded processors, but is also seen on some x86 processors [131].

Loki is able to emulate this behaviour with its two instruction inputs and *fill* instructions. The core can configure the L1 memory to send instructions to the channel of its instruction packet cache, and then execute a number of *fill* operations to populate the cache with useful instruction packets. (Recall that the *fill* instruction fetches an instruction packet to the L0, but does not execute it.) The L1 can then be reconfigured once more to send instructions to the core's instruction buffer so that the contents of the L0 are not disturbed.

This approach has a number of advantages:

- All of the level-0 cache can be used to store instructions for long periods, rather than needing to leave space to cycle through the remaining instructions.
- The instructions which are not pinned do not need to be written to the cache and read out again every time they are executed. Instead, they pass through the much cheaper instruction buffer which consumes less than 25% as much energy.
- In situations where it is difficult to predict which instruction packets should be pinned, it is possible to fall back on normal caching behaviour. The compiler-managed approach

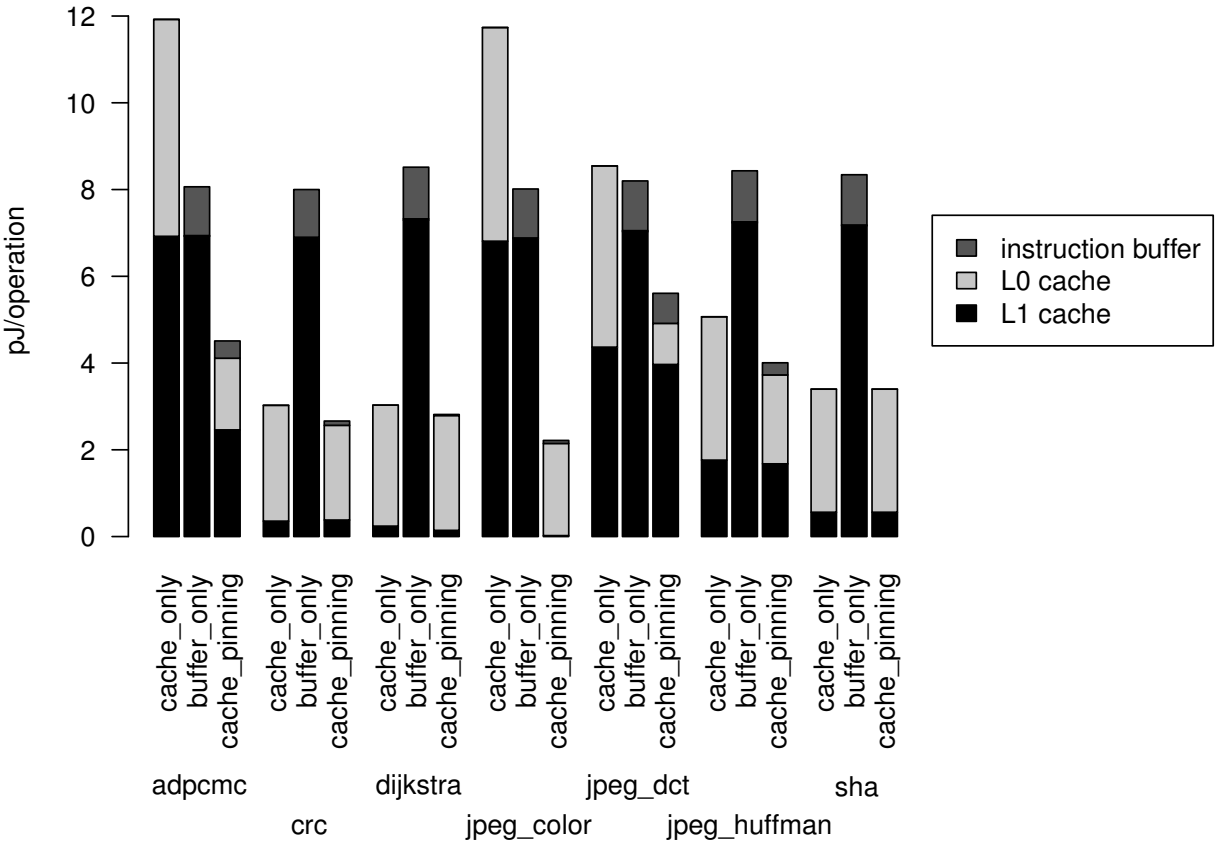


Figure 5.3: Energy distribution of instruction supply when using only the L0 cache, only the buffer and switching between the two using cache pinning.

described by Park et al. [102] eliminates cache tags to save energy, but this means that it is sometimes impossible to know what is already in the cache, forcing pessimistic re-fetching of instructions which may already be cached.

- Reduced compiler complexity – adding cache management to the Loki compiler is beyond the scope of this work, so it was useful to be able to hand-modify only frequently-executed loops.

Loki already has the instructions required to implement cache pinning: Park et al. make use of *fetch* and *jfetch* instructions on the Elm architecture, which have the same functionality as Loki's *fill* and *fetch* [102]. Park et al. also describe a way in which cache pinning can be applied automatically, using either static analysis or profile data. For this work, profile data was used to determine which regions of code are executed most often, and then the situation was treated as the knapsack problem: instruction packets have a "weight" corresponding to their size (rounded up to the nearest cache line) and a "value" corresponding to the number of L1 accesses which would be saved by storing the packet in the L0 (size of packet $\times$ number of executions), and the goal is to maximise the value of instructions stored in the cache.

Figure 5.3 shows how the instruction supply energy changes for a number of loops in the MiBench suite, and Figure 5.4 shows the overall effects of cache pinning. Loops were chosen based on their size: if they completely fit in the cache, then cache pinning would make no difference; if they were much larger than the cache, the effects of cache pinning would be smaller and harder to reason about.

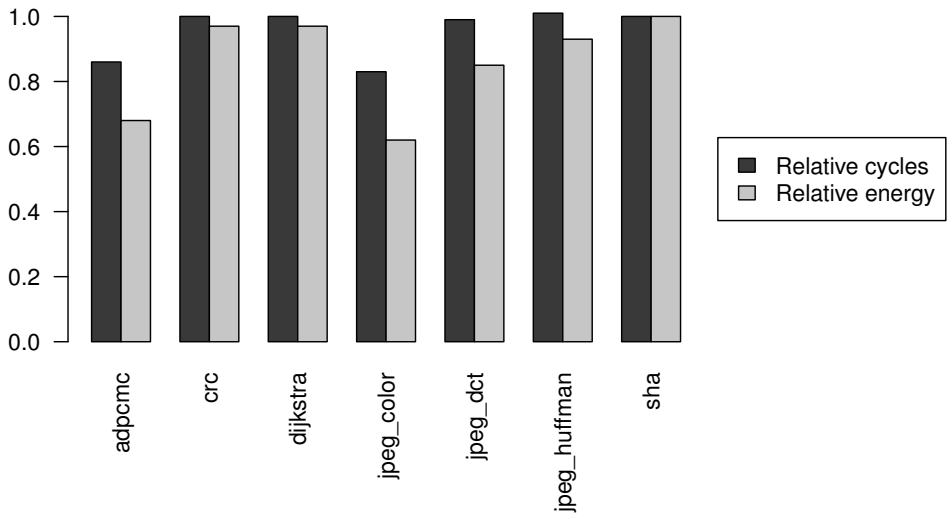


Figure 5.4: Effects of cache pinning, relative to the best single instruction source.

The data show that energy spent on instruction supply decreases for all benchmarks when using cache pinning. In some cases (*crc*, *dijkstra*, *sha*) the effect is small because the fully associative cache already does a good job of caching the most common path through the loop body. In other cases (*adpcmc*, *jpeg_color*), energy is greatly reduced as a large portion of the required instructions are now stored locally. Performance also improves in these cases as the average instruction fetch latency is reduced, and the L1 cache is active much less, so data requests can be serviced more quickly on average. *jpeg_dct* and *jpeg_huffman* lie somewhere between the two extremes.

Performance does not change by more than 1% in several cases. This is because in such regular regions of code, it is often possible to partially or completely hide memory latency by issuing a *fetch* in advance. This reduces the potential performance improvements provided by the lower-latency instruction packet cache.

The cycle count for *jpeg_huffman* increases by 1%. This is because the function is executed many times, but each time the pinned code is executed only a small number of times. Each time the function is called, it must first ensure that the cache contains the required instructions. This process requires reconfiguring the L1 cache to alter the instruction destination, which is relatively slow, taking tens of clock cycles.

The reconfiguration is slow in the current implementation because each memory bank updates its channel map table before sending the reconfiguration message on to the next bank – frequent reconfiguration was not considered when the L1 cache was designed. Often, cache pinning is used to improve the efficiency of a long-running loop, so these overheads become negligible. However, if the cost can be reduced, cache pinning becomes profitable more often, perhaps allowing shorter-running loops to be executed more efficiently.

There are two main ways these overheads could be reduced: make the memory bank reconfiguration faster, or remove the need for reconfiguration altogether.

Reconfiguration could be made faster by sending the configuration message to all memory banks simultaneously. This would require the addition of a new network which is capable of broadcasting data to memories. This would increase the energy and latency of the common case of point-to-point communication because additional multiplexers would be required to select which of the networks to receive data from, so the approach is undesirable.

Reconfiguration could be eliminated either by having the core steer all incoming instructions to the instruction store it wants to use, or by setting up two channels to memory at the beginning of the program, and having each one return instructions to a different input channel of the core. Each of these methods requires adding a new simple instruction to the instruction set. The instruction steering method needs an instruction to toggle which of the instruction stores is to be used. This is not very flexible because it removes the ability for a core to receive instructions from multiple sources simultaneously – for example, storing its own instructions in its cache, and receiving high-priority tasks from other cores at the instruction buffer. The multiple channel method needs to break the current restriction that all *fetch* requests are implicitly sent out on output channel 0. Adding an additional field into the instruction encoding would reduce the number of bits available for the immediate value, so the maximum size of relative jumps would be greatly reduced. Instead, I propose that an additional instruction is added which changes the output channel which is implicitly used for fetching instruction packets. This instruction would behave very similarly to the *rmtexecute* instruction: a single immediate argument which specifies an output channel to use. This similarity means that only very slight modifications need to be made to the pipeline (specifically, the decoder), and so the impact on the energy and latency of any modified components should be negligible.

Experiments found that the implementation of such an instruction is successful in eliminating the described overheads. *Set fetch channel* was implemented, and is described further in Appendix A.

5.1.4 Buffer pinning

Since Loki has two instruction stores which cost different amounts to access, it is possible to take the cache pinning concept further and implement buffer pinning. This is where as many instructions as possible are pinned in the cheaper instruction buffer, and any remainders stay in the L0 cache. This is useful for loops which are between the size of the buffer and the combined size of the cache and buffer; any larger and instructions will start getting replaced and need to be re-fetched.

In order to keep the design simple, I make the restriction that only a single instruction packet can be stored in the buffer at any time. This requires the addition of a single tag to the instruction buffer to record the address of the packet currently being stored. From a timing perspective, this addition fits easily into existing slack in the critical path, so does not force a more aggressive implementation for the buffer, and consumes negligible energy itself: 0.03pJ addition to the cost of every tag look-up, plus 0.6pJ whenever a new packet is fetched into the buffer (Table 4.10).

This technique is only beneficial if it is known that the instruction will be executed a certain number of times, as the contents of the buffer get overwritten as soon as a new instruction packet is fetched. Reading from the instruction buffer is approximately 2.1pJ cheaper than reading from the L0 cache, and bringing an instruction from the L1 cache into the buffer costs an average of 11pJ. Therefore, the instruction buffer will be the more energy-efficient option if the instruction will be executed six or more times before being evicted.

Since only a small number of benchmarks have main loops small enough to benefit from buffer pinning, individual case studies were performed on the *crc* and *sha* applications. Other benchmarks do contain loops of the right size, but they are not executed as frequently, and so a very long simulation would be required to generate enough data. Comparisons were made

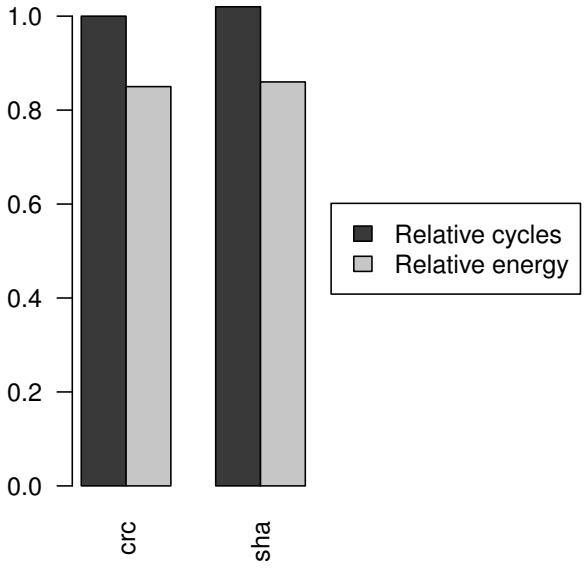


Figure 5.5: Behaviour of buffer pinning relative to cache pinning.

with cache pinning, as so far this was the best instruction supply method for each benchmark. Results are shown in Figure 5.5.

The inner loop of the *crc* benchmark fits entirely in the buffer, so the outer loop was stored in the L0 cache. There was almost no performance impact because almost all instructions fit in the cache previously, but energy reduced by 15% due to the use of the cheaper instruction buffer.

The *sha* benchmark is more complex. It consists of five tight loops (less than 30 instructions each) nested inside another loop. Each inner loop is already well-served by the instruction packet cache, but is evicted when the next inner loop is executed. To implement buffer pinning, each tight loop was cut into two instruction packets, if necessary: 16 instructions which fit in the buffer, and all remaining instructions which were stored in the L0 cache. These cuts introduced extra instructions to fetch the second halves of the loops and slowed execution by 2%, but allowed a large fraction of instructions (69%) to be read from the cheaper buffer, saving 57% of the pipeline's instruction supply energy (14% of the total energy consumed).

5.1.5 Summary

By giving each Loki core two instruction inputs, each consuming different amounts of energy, it is possible to tailor the instruction supply to suit each part of a program. A spectrum of approaches exist, depending on the size of the active code region. The smallest kernels can fit entirely in the instruction buffer and be supplied very efficiently, mid-sized kernels can use the L0 cache for additional capacity, and large code bodies return to the buffer to minimise overheads when neither structure is able to exploit locality.

Software management can be used to provide additional intermediate points on this spectrum: if the kernel is slightly too large to fit in either the buffer or the L0 cache, pinning can be used to guarantee that the most frequently executed instructions can be accessed cheaply.

Pinning did not increase energy consumption for any of the benchmarks tested, and gave savings of up to 40%. There was sometimes also a performance improvement because the higher-latency L1 cache was used less often.

Benchmark	Table size/bits	Description
adpcm	16×32	Table of constants
adpcm	89×32	Table of constants
bitcount	256×8	Bit counts for every possible byte
bitcount	7×32	Function pointers for various counting methods
crc	256×32	Table of constants
dijkstra	$nodes^2 \times 32$	Distances between all pairs of nodes
jpeg	64×32	8×8 block of pixels for processing
jpeg	64×16	Precomputed constants used in inverse DCT
jpeg	$2 \times 64 \times 8$	Two quantisation tables
jpeg	$2 \times 162 \times 8$	Two tables used for Huffman coding
sha	23×32	Data structure passed between loop iterations
stringsearch	256×32	Distance to skip if each character is seen

Table 5.2: Data structures found in MiBench programs.

5.2 Scratchpad

The software-managed scratchpad in each core can be used for a variety of purposes: it can store a frequently-accessed data structure, spilled register contents, or even a section of the stack. Only the first option is explored here, as the other two are best implemented by a compiler. It is possible that this restriction results in a suboptimal scratchpad selection, so future work will address the other two use cases and re-evaluate the decision.

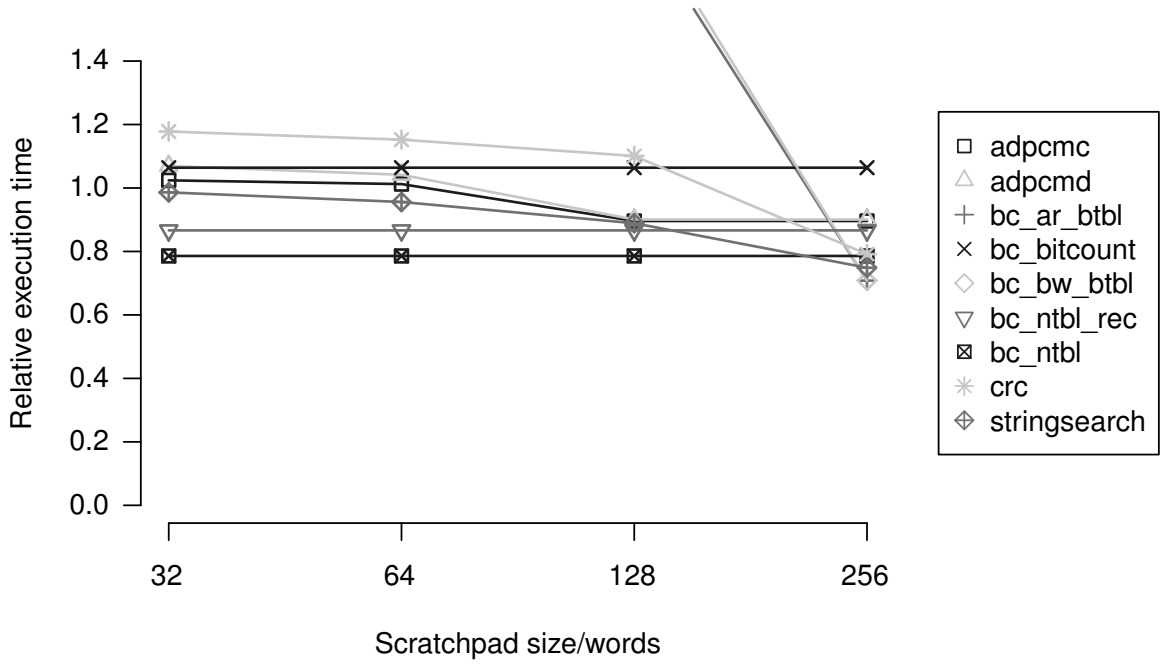
In this section, I explore the tradeoffs of different scratchpad sizes: larger scratchpads are able to hold more data, but are more expensive to access. Overprovisioning resources can therefore have negative effects on energy consumption, while underprovisioning results in additional instructions to determine whether the required data is in the scratchpad or L1 cache.

Accessing the scratchpad is much cheaper than the L1 cache, both because the structure is smaller and closer to the pipeline, and because fewer instructions are required. When an array of data is stored in the scratchpad, the array index is often exactly the same as the scratchpad address, so data can be read directly, whereas if the value is in memory, the array index needs to be converted into a memory address, and followed up with instructions to load and receive the data from the L1 cache.

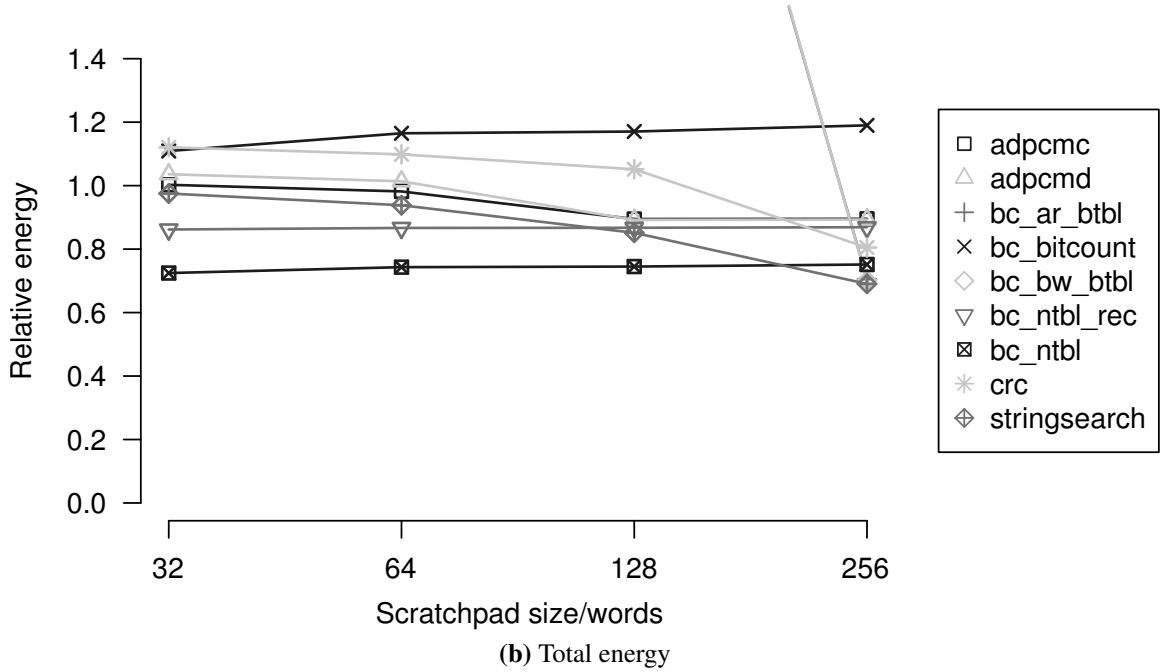
Table 5.2 shows a selection of the frequently-accessed tables and data structures from the benchmarks used. Most of the benchmarks make use of at least one array which would normally be stored in the cache hierarchy, but could be stored locally to reduce access costs. All except one of the data structures in Table 5.2 would fit in a core's scratchpad memory, if the largest one available is chosen (1kB). The exception is the adjacency matrix used in *dijkstra* whose size is data-dependent, and usually much larger than even the largest available scratchpad memory.

Benchmark source code was modified to fill the scratchpad with data when execution first started, and the *bitcount* benchmark was split into its independent counting functions. Benchmarks which do not make use of tables of data may still be able to take advantage of the local scratchpad to reduce the number of expensive memory accesses by using the techniques described above.

Figure 5.6 shows the execution behaviour when using various sizes of scratchpad, compared to the baseline case of no scratchpad at all. The 256-word scratchpad can be treated as the limit



(a) Performance



(b) Total energy

Figure 5.6: Behaviour of various sizes of scratchpad, relative to no scratchpad at all. All implementations are accessible within a single clock cycle.

case; it is the smallest structure into which all data fit for these benchmarks, so any larger scratchpads would only increase energy consumption.

In general, there is no improvement to either performance or energy except for the cases where almost the entire table fits in the scratchpad. This is because extra instructions are required to choose between the scratchpad and L1 cache, which negate the benefits of the shorter code path when the required data is stored locally. The reduced energy consumption of the scratchpad starts to outweigh the increased costs of executing more instructions only when around 75% of the table is stored locally.

Converting the scratchpad into a cache would avoid the need for separate code paths depending on if the data is stored locally or not, and so reduce the number of instructions required when a table does not fit. However, the addition of cache tags would add significant logic to the critical path and force a reduction in the size and efficiency of the data array. Benefits of the scratchpad such as predictable access times and minimal overheads would also be lost.

bc_ar_btbl and *bc_bw_btbl* are very similar and perform particularly poorly when the scratchpad is too small because the extra instructions mean that the loop body no longer fits in the cache, and must be re-fetched on every iteration. This effect could be greatly reduced using cache pinning (Section 5.1.3), by providing a larger scratchpad, or by packing multiple values into a single word. It was found that data-packing was worthwhile for these benchmarks: with a 32-word scratchpad, and 8 values packed into each word, execution time and total energy consumption both reduce by 15% over the baseline. This can be compared to the 29% saved when the scratchpad is large enough for data packing to be eliminated.

For those benchmarks which see improvements, performance improves by an average of 20% and overall energy consumption improves by 22%. 30-40% of the energy reduction is due to cheaper data access, and the remaining 60-70% is due to executing fewer instructions.

Since the scratchpad consumes such a small fraction of energy (1.5% average for 256 words) but can have such a large impact on performance and energy consumption, the largest possible data array which can be accessed in a single clock cycle is chosen: 256 words. This makes the scratchpad useful to as many benchmarks as possible, but may reduce any benefits for cases in which only a small amount of extra storage is required. The memory compiler is good at generating compact structures: the 256-word scratchpad is smaller than the 32-word standard cell register file, and consumes only around 10% of the total area of a core.

Scratchpads with multi-cycle access times were not explored, as the benefits over the L1 cache diminish significantly, and because in the set of benchmarks used, there was little need for anything larger.

The chosen scratchpad costs an average of 3.65pJ to read, and 3.44pJ to write, compared with 0.65pJ and 0.95pJ for the register file and 16.5pJ and 16.0pJ for the L1 cache (including network costs). Values copied from the L1 cache into the scratchpad need to be accessed an average of 1.6 times each before energy is saved compared to accessing the L1 every time. This suggests that it may be beneficial to use the scratchpad for even quite short-lived values. It is not beneficial to use a larger register file for these short-lived values because of the energy impact on the common case, and because an extra bit would be required for each register in the instruction encoding, reducing the bits available for immediate values and other purposes.

Figure 5.7 shows how data supply energy changes when the selected scratchpad is used, compared with no scratchpad. In all cases but one, the scratchpad reduces energy consumption through a combination of storing data in cheaper structures, and reducing the total amount of data which needs to be manipulated by eliminating instructions which generate memory addresses. The outlier is *bc_bitcount*. This function makes use of many large literals, but storing

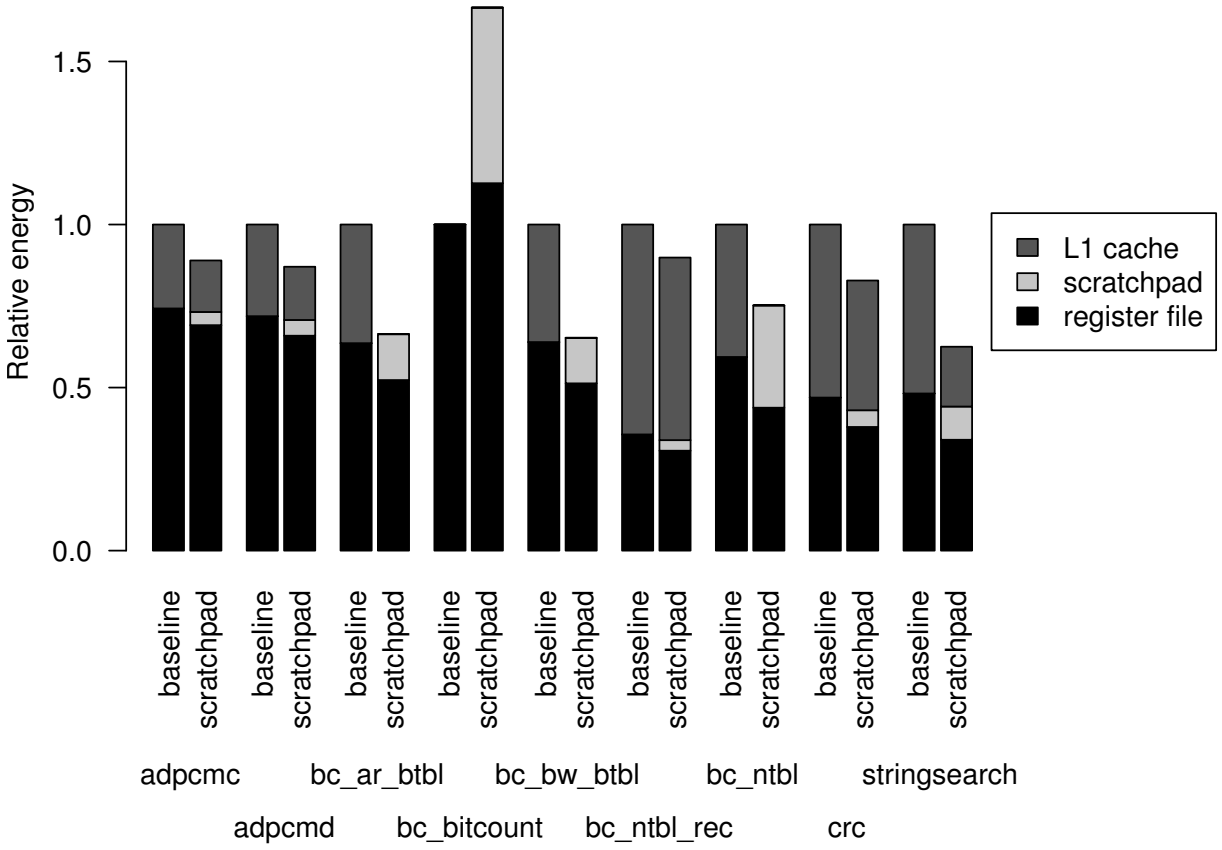


Figure 5.7: Data supply energy distribution with and without use of the scratchpad.

them in the scratchpad prevented the compiler from performing optimisations which depended on the literal values. This increased the code size, with a corresponding increase in energy.

Data supply costs can also be reduced using optimisations to the register file. These optimisations are not explored here, as the register file currently consumes only a small portion of the total energy. Possible optimisations include:

- providing narrow register file ports, allowing the common case of small values to be read more efficiently;
- explicit operand forwarding, where results which will be immediately consumed are not written back to the register file [16];
- skewing the multiplexer tree or using a separate register file to create a subset of cheaper registers for short-lived or many-access values [42, 134];
- reducing the number of ports on the register file – the addition of explicit operand forwarding and network communication may mean that the performance impact of removing a port is acceptable.

5.3 Network

Each tile of the Loki architecture contains a number of different networks, optimised for different use cases. Cores and memory banks communicate via fast crossbars in order to keep memory latency down; cores communicate with each other via slower crossbars which are capable

of multicast, to allow them to work together more efficiently; and memory banks communicate with each other using a simple ring network.

This section explores different implementations of the networks connecting cores and memory banks, and the number of components connected to those networks. The core-to-core network is explored in Chapter 6, when multiple cores are used by benchmarks and communication between them is required, and the memory-to-memory network is beyond the scope of this work.

When using only a single core, network buffers do not limit execution at all, so it would be unfair to explore different implementations at this stage. An investigation is performed in Chapter 6, where multiple cores are active simultaneously, so buffers experience higher activity and have a larger effect on performance and energy consumption. The number of network channels available also remains constant: changing the number of register mapped inputs would require changes to the compiler's register allocation, and changing the number of output channels would require changes to the instruction encoding.

Multicast is a useful feature of a network as it can allow higher-radix communication, but its addition generally results in a slower interconnect as the long wires must be able to drive more outputs with a higher total capacitance. The number of components on a tile is also a factor: a larger number of memory banks allows more information to be stored locally, and cheaper communication between more cores allows more effective parallelism. However, the complexity of crossbar networks increases quadratically with the number of connections, suggesting that fewer components may be more efficient.

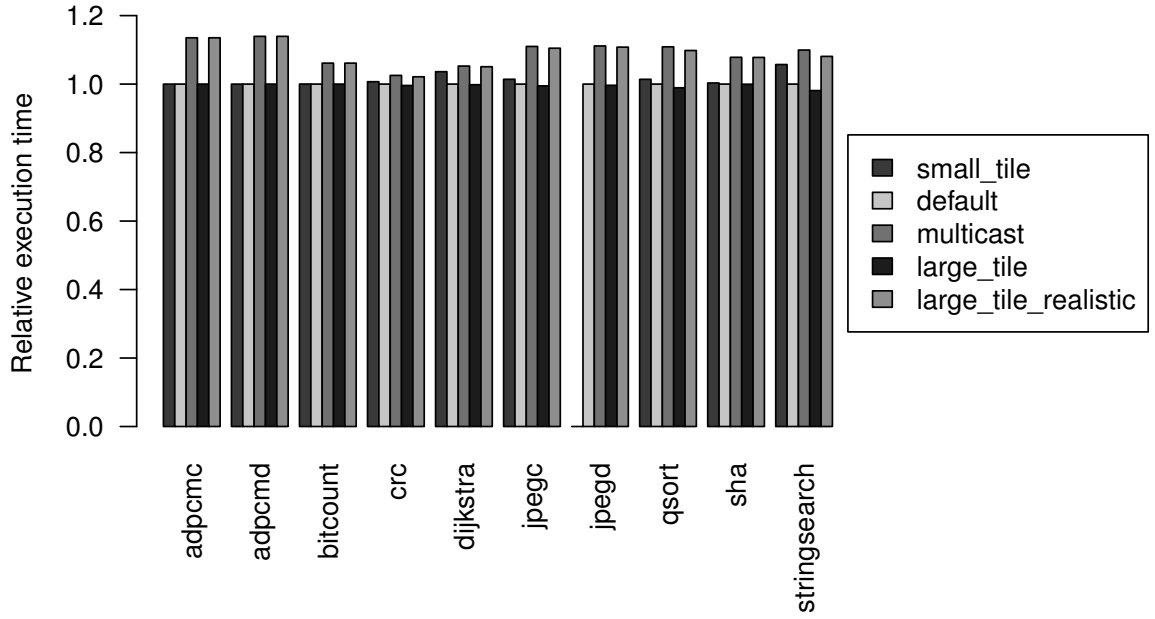
Figure 5.8 presents results relative to a *default* crossbar network connecting 8 cores and 8 memory banks with a half-cycle latency.

The *small_tile* implementation connects 4 cores and 4 memory banks using a crossbar with a half-cycle latency. With this smaller tile, execution is an average of 1% slower, showing that memory capacity is not a limitation for these benchmarks, and energy is an average of 2% lower.

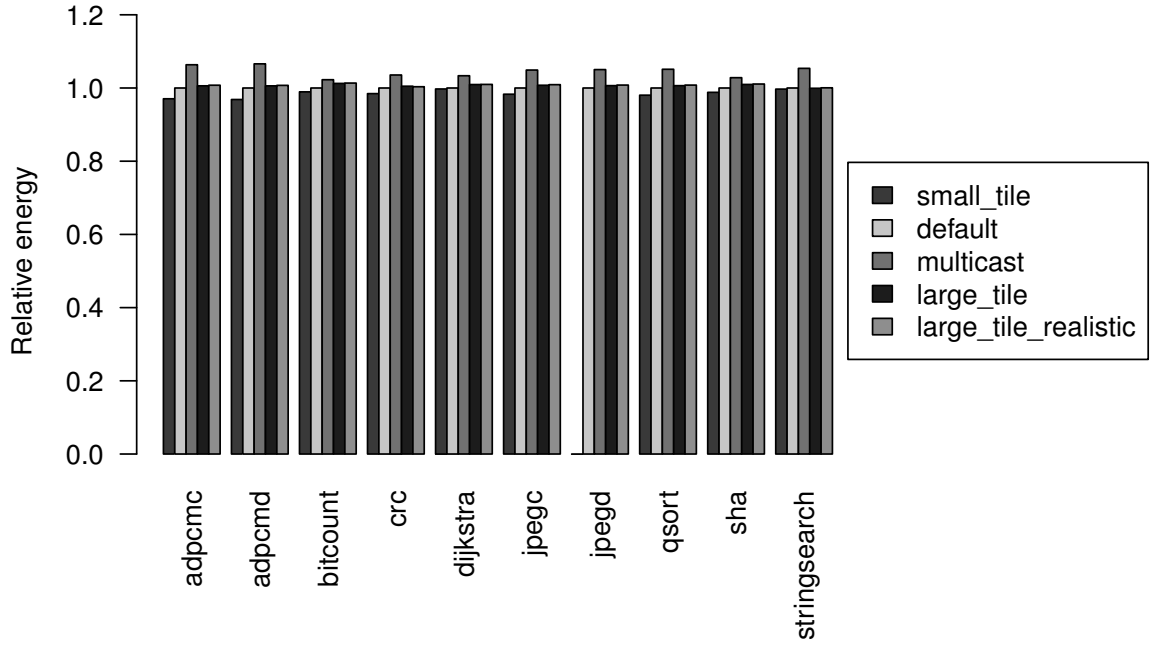
Conversely, a *large_tile* with 16 cores and 16 memory banks observes a 1% increase in energy consumption. This is an impressively small increase, considering the quadratic complexity of crossbars and the frequency that the network is used. If the crossbar is able to transfer data within half a clock cycle, keeping memory latency under two cycles, a small performance improvement of less than 1% is seen due to the increased memory capacity. In practice, however, such a network requires at least one full clock cycle to transfer data, resulting in a 9% increase in execution time.

Finally, a crossbar capable of multicast was explored, though none of the multicast features were used in this experiment. The latency of the network was one clock cycle. Energy increased by 5% as long wires which span the whole tile are used even when communicating with a nearby component, and execution time again increased by 9% due to the increase in memory latency.

The figures show that the default implementation is a sensible one: increasing the size of the tile, or using a more-complex network increases the network latency, which in turn has an impact on memory latency, and reduces performance. It seems unlikely that the benefits of multicast from memory banks to cores will outweigh the costs; it is proposed that when multicast behaviour is required, a single core should load a value, and use the core-to-core multicast network to distribute it to all other destinations. Multicasting values *to* memory banks (for example, to write the same value to many locations at once) could be useful, and is the subject of future work.



(a) Relative execution time



(b) Relative energy consumption

Figure 5.8: Comparison between different sizes and implementations of networks between cores and memory banks. All figures are relative to the *default* implementation of 8 cores and 8 memory banks connected by a fast crossbar. A *small* tile contains 4 cores and 4 memory banks, and a *large* tile contains 16 of each. The *multicast* and *realistic* implementations have a one-cycle latency, and all others are able to transfer data within half a clock cycle.

Component	Entries	Data width/bits
Channel map table	16	32
Instruction buffer	16	32
L0 data array	64	32
L0 tags	16	30
L1 data array	8192	32
Network buffers	4	32
Register file	32	32
Scratchpad	256	32

Table 5.3: Sizes of components in the Loki architecture.

There is little difference between the default tile and one half its size, so the larger option is selected to allow more exploration of parallelism in the following chapter, and to increase memory capacity for those applications which may require it.

5.4 Summary

Table 5.3 summarises the module implementations selected for the baseline Loki architecture. Eight cores and eight 8kB memory banks make up each tile, and they communicate via a fast point-to-point crossbar network. The sizes of the channel map table and register file are fixed by Loki's instruction encoding, and were selected as sensible starting points. As mentioned previously, network buffers will be investigated in Chapter 6, where multiple cores are active simultaneously, and buffers have a larger impact on performance and energy consumption.

Table 5.4 gives a summary of typical energy consumptions for common operations. All assumptions are based on average figures observed in execution traces.

Figure 5.9 shows a breakdown of which components consume the energy when executing the baseline MiBench applications; the instruction buffer and scratchpad are not used. Components are grouped as follows:

- *data supply* consists of the register file and the scratchpad
- *instruction supply* consists of the instruction packet cache, the instruction buffer, and all tags
- *network* consists of all networks and arbitration
- *network interface* consists of the channel map table and network buffers

Energy per operation ranges from 10.0pJ for *bitcount* to 20.6pJ for *adpcmd*, and is highly dependent on the performance of the cache. In cases where the cache performs well, less than 3pJ is spent per operation on supplying instructions, as the main action performed is an L0 read. When the cache performs poorly, each instruction must be read from the L1 and written to the L0 before being read, at a total cost of almost 12pJ. Effective use of the instruction buffer as described in Section 5.1 can be used to reduce this, as can the use of multiple cores (Chapter 6).

Figure 5.10 shows the areas of all modelled components; it is possible to fit eight cores, 64kB memory and all required interconnect into 1mm^2. Some parts of the pipeline consume a disproportionate amount of area relative to the frequency they are used – the cache tags make

Action	Assumptions	Energy/pJ
IPK cache read	-	2.5
IPK cache write	-	2.4
Instruction buffer read	-	0.35
Instruction buffer write	-	0.75
Tag look-up (cache + buffer)	5 bits toggle	0.69
L1 instruction read	-	6.9
Pipeline register write+read (32 bit)	12 bits toggle	0.28
Instruction decode	-	0.52
Register file read	8 bits set	0.65
Register file write	7 bits toggle	0.93
Scratchpad read	-	3.7
Scratchpad write	-	3.4
L1 data read	Read 32-bit value	9.0
L1 data write	Write 32-bit value	12.0
32-bit bitwise operation	7 bits toggle in operand 1, 5 bits toggle in	1.6
32-bit addition	operand 2, 7 bits toggle in output	1.9
32-bit multiplication	Compute lower 32 bits of result	8.9
0.5mm 32-bit bus	12 bits toggle	0.34
Crossbar interconnect	12 bits toggle, average distance is 0.5mm	1.3
Multicast interconnect	12 data bits toggle, 1 bitmask bit toggles	1.0
Network buffer read	-	0.40
Network buffer write	-	0.68
Arbitration	-	0.45
Total cost of L1 load (including network)	Data stops in all network buffers on the way	16.5

Table 5.4: Average energy consumption of common operations in a 40nm low-power process. Assumptions of bit switching are based on average figures from experimental data.

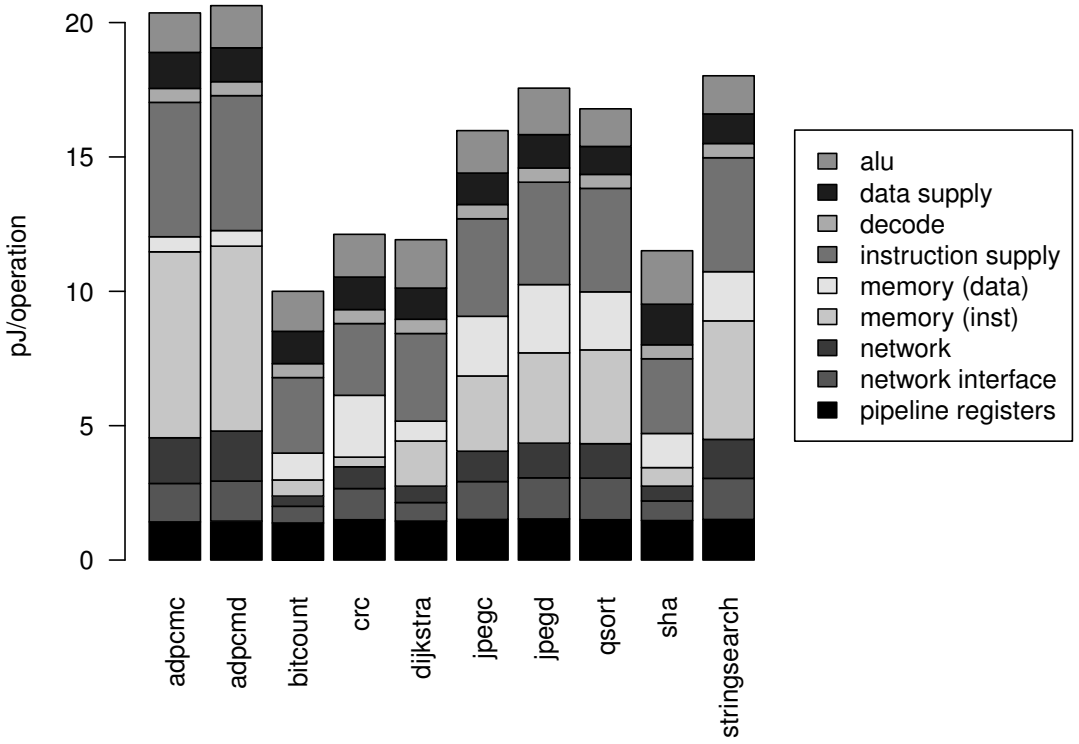
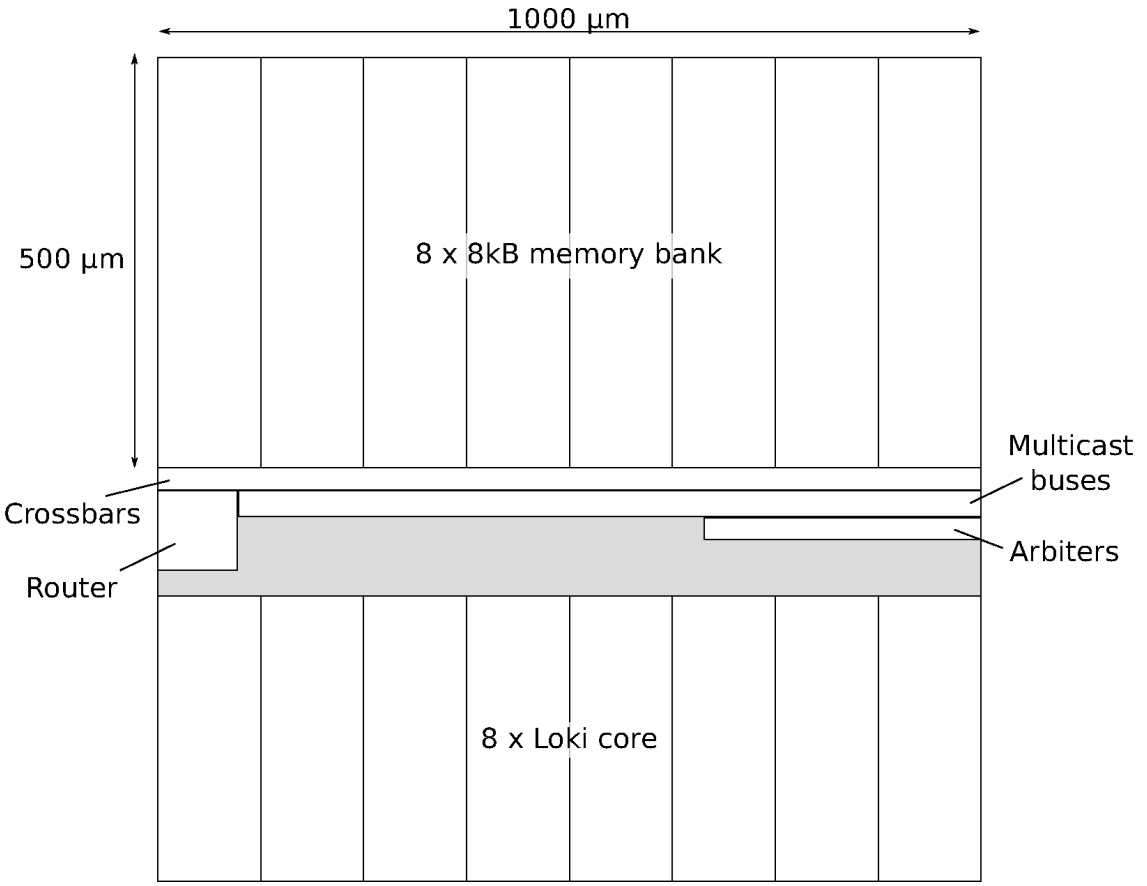


Figure 5.9: Distribution of energy consumption for the baseline Loki architecture.

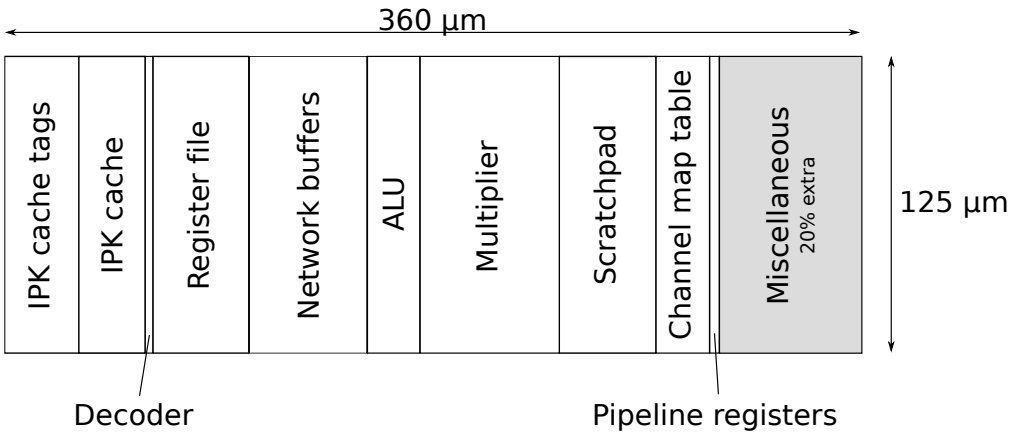
up 9% and the multiplier takes up 17% of the total core area. Reducing the area consumed by these modules would help bring other components closer together and reduce communication costs, but these effects are expected to be relatively small compared to other savings, so are not explored here.

I observe that instruction supply dominates the total energy, as was found by Dally et al. for a typical embedded processor [33]. Cache pinning (Section 5.1.3) can be used to improve this by up to 40% in some cases, and the following chapter explores several ways in which multiple cores can work together to reduce energy.

I now derive an estimate of the number of Loki cores which could be active simultaneously in an embedded system. I assume a typical mobile phone TDP of 2W, including communication off-chip. For a future system with 25GB/s of off-chip memory bandwidth (NVIDIA's Tegra 3 has up to 6.4GB/s [120]), where each bit communicated costs 5pJ [17, 123], off-chip communication will cost a maximum of 1W. The introduction of chip stacking and through-silicon vias (TSVs) will reduce these communication costs by a factor of roughly 1000, though memory bandwidth will likely increase greatly as a result [19]. This leaves 1W for the cores themselves. I assume that each core is able to achieve on the order of 10pJ per operation at 435MHz, with an average IPC of 0.72 (Section 4.3.1).



(a) Tile



(b) Core

Figure 5.10: Floorplans using results of synthesis. Shaded regions do not contain any modelled components, but are retained to cover any inaccuracies.

$$power = \frac{energy}{time} = \frac{energy}{instructions} \times \frac{instructions}{cycles} \times \frac{cycles}{time}$$

$$= \frac{energy}{instructions} \times cores \times single\ core\ IPC \times clock\ frequency$$

$$1W = 10pJ \times cores \times 0.72 \times 435MHz$$

$$cores = 319$$

319 cores corresponds to a total of 40 tiles, consuming an area of 40mm^2 with a total of 2.5MB of cache. For comparison, the Tegra 3 system-on-chip occupies 80mm^2 in a 40nm process, including additional logic such as I/O pads and memory controllers. Of course, it would be possible to place more Loki cores on a chip, but it would not be possible to use them all simultaneously due to the power constraints. This is not necessarily a bad thing – it would allow further specialisation of those cores which were in use. Alternatively, "spare" area could be used to increase cache capacity; some tiled architectures such as Elm [14] provide tiles which contain only cache.

5.5 Comparison with other work

When so much of the infrastructure used is new (microarchitecture, instruction set, compiler), it is difficult to form fair comparisons with other work. Many other research groups developing novel architectures have also encountered this problem. For this reason, I have provided a lot of raw numbers, along with estimates of their error. I attempt to make comparisons as fair as possible by using standard source code for benchmarks, and making notes of various intermediate results such as dynamic instruction count to help compare compilers and instruction sets.

A comparison was made with an ARM1176 processor, synthesised in the same 40nm technology, and running at 700MHz. Energy figures were scaled from published results at 65nm [10] and confirmed by direct power measurements from a development board using a power monitor [89], giving a figure of approximately 140pJ/operation.

In order to further reduce the effects of system calls, all print statements and file accesses were removed from benchmarks and all program inputs were embedded into the executables. Hardware performance counters on the ARM were used to measure instructions executed and clock cycles. Each benchmark was executed a large number of times under Debian Wheezy, and the shortest time was selected. Executing benchmarks bare-metal was not found to make a significant difference.

Figure 5.11 compares ARM and Loki versions of the benchmarks. When using identical source code, only four of the benchmarks were able to compile on Loki, but comparisons with less-modified versions of the benchmarks suggest that these results are representative. Dynamic instruction counts are 1.4-2.2× higher on Loki at present. There is no single dominant reason for this, but rather several smaller features in the ARM instruction set which Loki does not offer, including:

- A barrel shifter which allows virtually all shifts and rotations to be folded into other instructions at no cost.

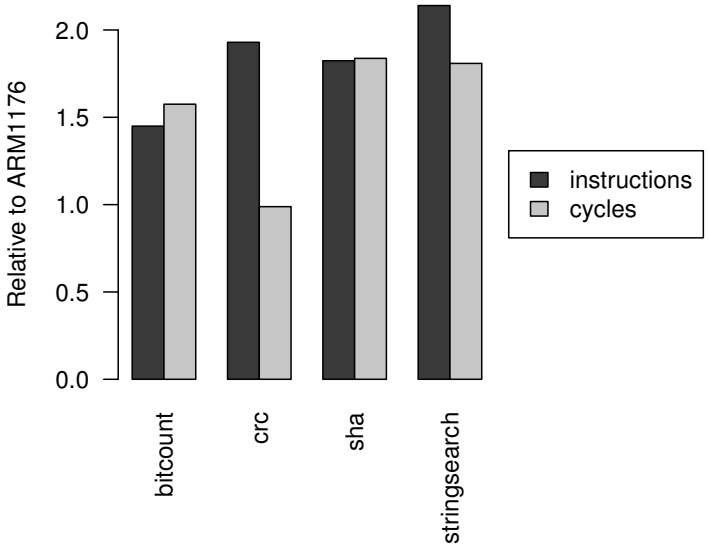


Figure 5.11: Comparison with ARM1176. Identical source code was used for benchmarks on the two platforms.

- A richer set of condition codes upon which execution can be predicated.
- Instructions to extract and sign-extend subsections of data words.
- More memory addressing modes, particularly register+register.
- *Load/store multiple* instructions which allow multiple values to be transferred to/from memory and are useful when working with the stack.

The final three points could be added to Loki relatively easily through the introduction of new instructions. The first two would be more complicated, requiring modifications to the instruction set to encode the extra information. Such optimisation of the instruction set is left for future work; for this work, it is sufficient to know that behaviour of the Loki baseline is comparable to other architectures.

Overall execution on a single Loki core is 1-1.8$\times$ slower than the ARM core clocked at the same frequency, showing that Loki is executing a similar number of instructions per clock cycle as the ARM processor. (Loki has better IPC in *crc* because the ARM code is more efficient and has fewer instructions, but both architectures are memory-bound and take the same amount of time to execute the program.) This suggests that as the compiler and instruction set improve and reduce the number of instructions executed, performance will get closer to that of the ARM. Meanwhile, Loki can make use of parallelism to match the ARM core's performance, and is able to execute 7-14 instructions for each ARM instruction while still having a lower energy consumption.

The ARM processor occupies approximately 1mm^2 on die, including 32kB of cache and a double-precision floating point unit. (A 128kB L2 cache is also available, but is not included in these area estimates.) With the same resources, Loki is able to provide 8 cores, each with 50-100% the integer performance of the ARM processor, and twice as much cache. With all eight cores operational, total power consumption would be comparable to the ARM processor, but the work being done should be significantly greater. Loki is unlikely to be able to compete on

floating point performance, however; providing efficient support for floating point computations is a subject of future work.

5.6 Conclusion

In this chapter, I have determined a sensible baseline implementation for a tile of the Loki architecture. All decisions were made based on applications running on a single core, so the next chapter combines multiple cores in a variety of different ways to stress other parts of the design and refine it further. Additional tweaks may be required as the architecture is used in ways not explored in these benchmarks, such as large applications covering multiple tiles. It is expected that the selected design is a good base upon which these modifications can be made.

The Loki architecture is able to achieve an energy efficiency of 10-20pJ per operation for single-core applications, which compares very well with other embedded architectures. Performance is lower, but cores are designed to work together to overcome this handicap. There is scope to execute many Loki instructions for the same energy as a single ARM instruction, suggesting that when parallelism is available, Loki will be able to match or exceed the performance of other embedded processors while still consuming significantly less energy.

Cache and buffer pinning were shown to be effective on the Loki architecture, saving up to 40% of total energy consumed by an application. It is anticipated that as programs are spread across an increasing number of cores, their tasks will become increasingly specialised, and many cores will be able to make effective use of cache or buffer pinning. This improvement in caching behaviour will increase both performance and energy efficiency.

CHAPTER 6

EXPLOITING TIGHTLY-COUPLED CORES

Virtually all applications exhibit some form of parallelism, but the type and amount varies from application to application, and often between phases within a single program [101]. One of the aims with Loki is to support as many different types of parallelism as possible, at minimum cost to sequential execution. This allows the widest possible range of applications to be parallelised, and so avoids the need to resort to a more-complex co-processor for executing sequential code. If a co-processor was required, it would reduce the number of Loki cores that could fit on a chip (reducing potential performance), and it would often be difficult to achieve an energy efficiency comparable to the Loki cores. Support for many forms of parallelism can also increase the amount of parallelism which can be exploited within each application, particularly for applications with parallelism which would be difficult to access with traditional architectures. This increases performance and helps justify the provision of such a large number of processing units.

Simply having cheap communication (in both time and energy) allows a variety of low-level parallel building blocks to be implemented using multiple cores: some of the possibilities and a number of optimisations are explored in this chapter. I call these building blocks *execution patterns* – this is meant to imply that there is no hardware reconfiguration involved in switching from one pattern to another, and that multiple patterns can be combined to produce something more complex. Each pattern is just a different way of mapping parallel software onto multiple cores. These execution patterns allow a *virtual architecture* to be built for each program which most naturally takes advantage of the parallelism available; there is no need to restructure the program's code to take advantage of the architecture, as is often necessary when writing multithreaded programs today. Having a wide range of low-level parallel building blocks makes parallelisation easier, reducing the costs of parallelising a program, and also increasing the amount of parallelism which can be extracted easily.

Loki attempts to overcome limitations imposed by Amdahl's law by allowing multiple cores to form a single virtual processor, and work together to execute a single thread. It is therefore possible to achieve both high parallelism and high sequential performance. It has previously been suggested that this dynamic approach is better than any static mix of weak and powerful processors [53].

The goal of this chapter is not to find the most efficient way of implementing each execution pattern, but to demonstrate that a wide variety of parallel execution structures are made effective by cheap communication. There is plenty of scope for optimising the execution patterns, both with hardware modifications and improved software. Parallelism is explored only within a single tile of 8 cores, and the possibilities of extending execution patterns to use multiple tiles are discussed where appropriate.

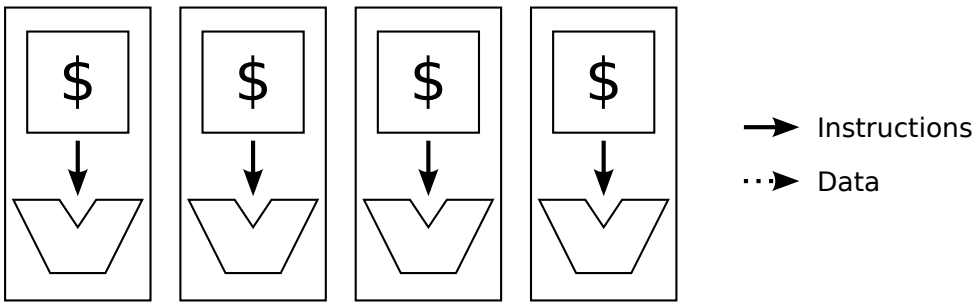


Figure 6.1: MIMD execution pattern. Each core independently issues instructions from its local cache, with instructions potentially coming from separate applications.

Due to the specialisation inherent in execution patterns, only a subset of benchmarks can effectively use each one; conversely, each benchmark is only able to take advantage of a subset of execution patterns. There is some scope for automatically determining when an execution pattern applies, but for the moment, programmer assistance is often likely to be more fruitful. Discussion of how each type of parallelism could be exposed to the compiler is included in each section.

6.1 MIMD

The multiple instruction, multiple data (MIMD) paradigm usually consists of running multiple independent threads at the same time (Figure 6.1). For the purposes of this discussion, I expand this definition to include multiple independent applications executing concurrently.

This is the type of parallelism most-frequently exploited by multicore architectures since no program analysis is required, but the amount of parallelism is often quite low because few applications need to perform computation simultaneously. In order to prevent the different applications from interfering with each other, a solution to the address binding problem is required. For most modern hardware, virtual memory is used, though it is also possible to choose memory addresses at load- or compile-time. Loki does not yet support any of these solutions, and this thesis focuses on other, less common forms of parallelism, so this section discusses other features which will help once an implementation has been selected.

Loki's configurable memory system allows each process to be given its own virtual cache made up of a fixed number of L1 cache banks. This would help to make memory behaviour more predictable by removing the influence of other processes. However, providing a fixed amount of memory may be detrimental: some parts of the program may have larger working sets, and some parts may leave much of the provisioned resources unused. Allowing memory reconfiguration allows the best approach for each situation to be selected.

Loki's channel map table also gives the opportunity to transparently move threads around the chip at runtime. This can be used, for example, to physically move computation near to the off-chip memory interface when using lots of data, or away from other threads to avoid network contention.

6.2 Data-level parallelism

The data-level parallelism (DLP) execution pattern involves multiple cores executing the same code, but producing different parts of the result. This technique is most often used to paral-

lelise a loop: if the iterations of a loop are independent of each other, they can be executed concurrently. If the iterations of a loop are not independent, DLP often becomes more difficult to exploit because there must be communication and synchronisation between the different iterations. With the expensive communication mechanisms of traditional architectures, this can often mean that parallelising the loop is no longer profitable. Speedup is limited by the length of *sequential* regions: parts of the loop body which cannot execute out of order between loop iterations. There are some techniques which transform the code to minimise the impact of inter-iteration dependencies by shortening sequential regions [29], but there are some loops which still are not worth parallelising. Loki's very cheap communication means that DLP is profitable more of the time.

Much of the execution time of typical programs is spent in loops [36], so if these loops can be parallelised, large gains can be made. Data-level parallelism is often more scalable than other forms such as instruction-level or thread-level parallelism, as it is limited by the number of loop iterations (often a large number), rather than the low-level data dependencies or structure of the code. Indeed, supercomputers are built to exploit huge amounts of data-level parallelism, and 70% of 2011's 20 "greenest" super computers make use of architectures optimised for DLP execution such as GPUs and the Cell processor [4].

DLP execution can either be decoupled, where different cores execute the same code, but at their own pace, or in lockstep, where each functional unit executes the same instruction at the same time. Architectures capable of using a single instruction to perform many copies of the same operation simultaneously are typically referred to as single instruction multiple data (SIMD). SIMD execution can be much more efficient than decoupled data-level parallelism because each instruction only needs to be fetched and decoded once before being issued to many functional units. This approach can be restrictive, however, in the case of divergent control flow within a loop. If there is divergent control flow, then different processing lanes will need to execute different instructions, so the lockstep mechanism will no longer be appropriate. Some architectures solve this problem by having a bitmask, with one bit for each lane, representing which branch was taken by each [40]. Instructions can then be issued in lockstep to one subset of cores and then the other. This requires that both sides of the branch are executed sequentially, which increases the length of the critical path, and the technique becomes increasingly complex if non-trivial control flow is required, but it does allow the efficient lockstep mechanism to be used more of the time.

Many modern processors have some DLP capability: instructions are provided which treat register contents as multiple independent values. This approach is called SIMD within a register (SWAR) and allows parallelism without the need for extra hardware; only small changes to existing functional units are required. For example, a 32-bit value can be treated as a vector of four separate 8-bit values. In order to add two of these vectors together, the only hardware modification required is the ability to break the carry chain in the adder to make each addition independent. This technique has such potential that there are now a number of instruction set extensions which specify increasingly wide registers and wide functional units, capable of large numbers of simultaneous operations – an example is the Advanced Vector Extension (AVX) which specifies 16 256-bit registers [58]. These extra features can be very useful [121], but are often difficult to exploit, and parallelism is limited by the hardware available. Loki provides very simple support for up to 8-way DLP groups within a tile, and it is then possible to replicate these groups as many times as desired to increase parallelism.

Graphical processing units (GPUs) are designed to execute data parallel code quickly, as there are many processes in graphics which perform independent computation for each pixel

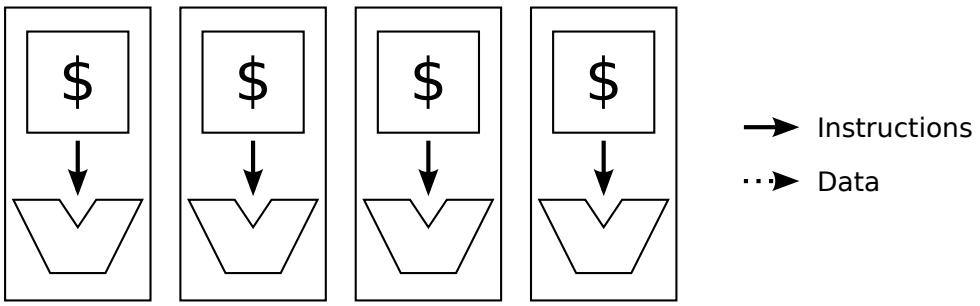


Figure 6.2: DLP execution pattern. Each core independently issues instructions from the same application from its local cache.

or each vertex of a model. Their architectures are specialised by allowing control costs (such as fetching instructions) to be amortised across multiple processing elements, and by using a huge number of threads to hide memory latency and reduce the need for on-chip caches. This specialisation makes GPUs much more energy-efficient than more general-purpose processors, but limits the types of application that can be executed effectively.

This section first explores mapping decoupled DLP execution to the Loki architecture, as it is more flexible than SIMD and can be used in more cases, allowing more parallelisation opportunities. I then explore reducing redundancy by extracting common tasks to a *helper core* in Section 6.2.3. Finally, hardware modifications to allow the more-efficient SIMD execution are explored in Section 6.2.4.

6.2.1 DOALL and DOACROSS

When all iterations of a loop are independent (known as DOALL parallelism), executing them in parallel is trivial; the iterations can be sliced in whichever way is most convenient, and distributed across the available functional units.

For these experiments, data-level parallelism is implemented on Loki by telling multiple cores to fetch the same instructions, and updating the way the iteration counter is managed (Figure 6.2). The iterations are striped across the cores: if there are 8 cores, then core 0 will execute iterations 0, 8, 16, etc. This was done for two reasons:

1. It is very simple to determine when each core should stop. Each core keeps an iteration counter, and increments it by the number of cores after each iteration. The core stops work when its counter is greater than or equal to the total number of iterations to be carried out. If each core is to be given a sequential block of iterations to execute, division operations would be required to determine the block boundaries, which are expensive on architectures such as Loki which have no dedicated division logic.

2. The parallelism is very fine-grained, which means minimal changes are needed to support cross-iteration dependencies (described below). If a core executed a larger group of sequential iterations, they would all have to complete before a result could be passed on to the next core, allowing it to begin. This approach would be very close to a completely sequential implementation.

In cases where there is locality between neighbouring loop iterations, it may be preferable to instead divide the iteration space into contiguous blocks.

When there are fixed cross-iteration dependencies (DOACROSS parallelism), it is necessary to set up communication channels before the loop begins, and modify the loop body so that it

sends and receives data over the network when appropriate. On Loki, this can usually be done with zero performance overhead, as reading from the network simply replaces a register read, and sending onto the network is an optional feature of most instructions. The only time there is an overhead is when a value received from the network is needed by multiple instructions: since reads from network buffers are destructive, the value cannot be read twice, so must first be copied into a register. There also needs to be a little work sending the initial live-ins before the first iteration, and draining any superfluous values after the loop completes.

More complex dependency patterns can also be supported – these would require a new channel to be set-up on each iteration. Careful synchronisation would be necessary to ensure that no two senders write to the same channel at the same time and that there is no possibility of deadlock.

6.2.2 Evaluation

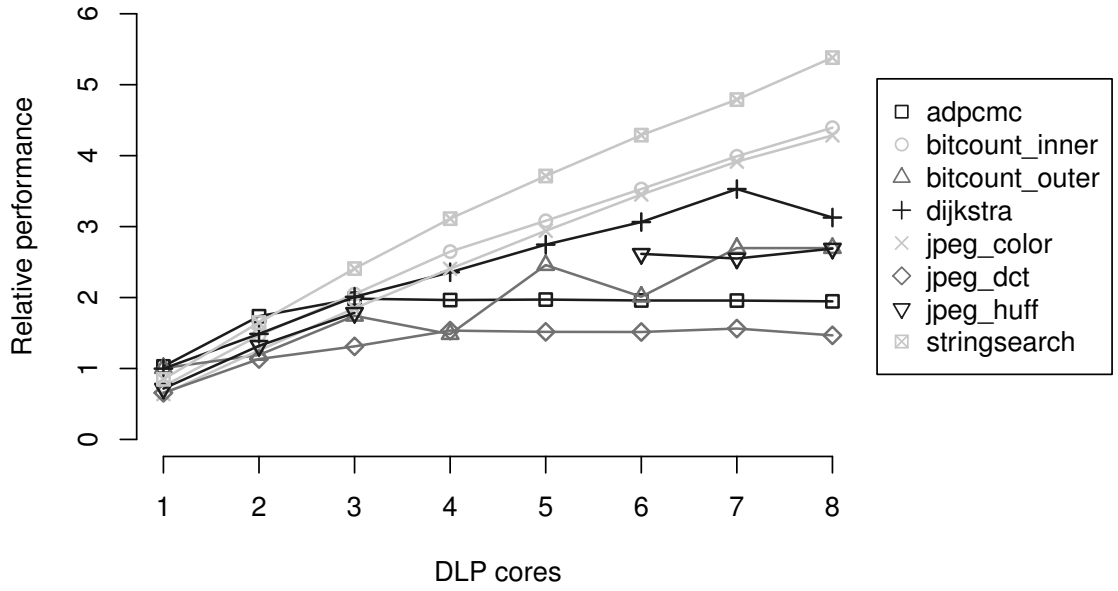
In order to evaluate the impact of data-level parallelism, a number of benchmarks were modified at the source code level. Modifications included initialisation of the additional cores, and production of a final result from each core's partial result. Each core was given its own stack space, but the heap was shared. The stacks were positioned in memory such that they mapped to different cache banks, to minimise conflicts. The scope of some variables was also changed depending on whether they should be shared between all cores, or private to each core.

A number of loops exhibiting data-level parallelism were selected from MiBench, and data was collected only while the loops were executing. The *adpcmc* benchmark contains DOACROSS parallelism, while all others contain DOALL parallelism. *bitcount* has multiple nested loops, so parallelism was explored at two levels.

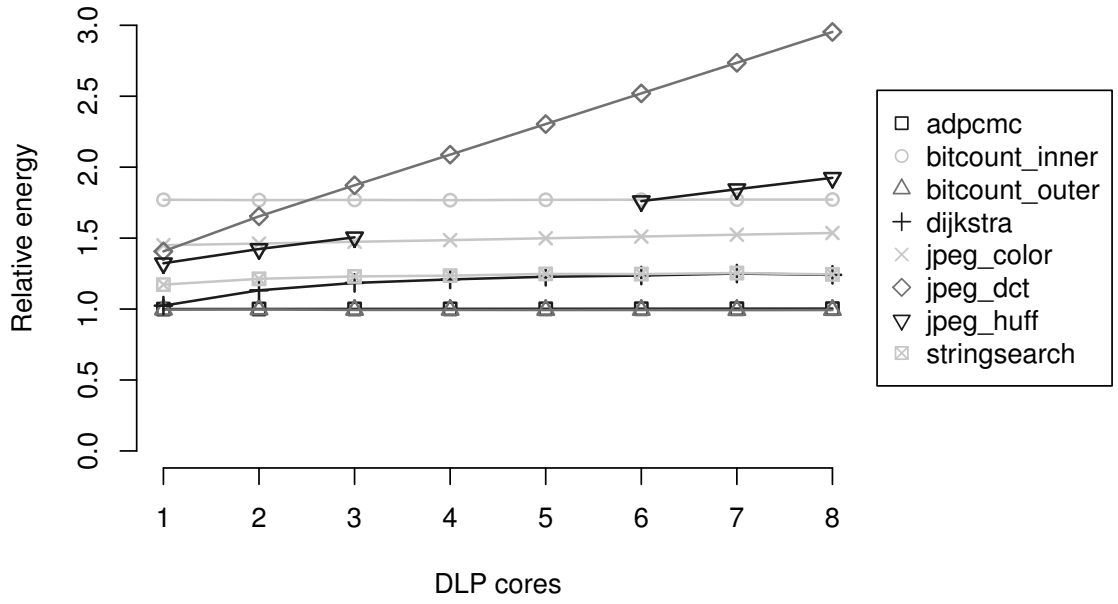
Figure 6.3 shows how performance and energy scale as the number of cores used increases; all benchmarks are compared to the baseline single-core implementation. The loops display a wide range of behaviours: some, such as *stringsearch* scale well, achieving a 5.4× speedup on 8 cores, and others such as *jpeg_dct* do not scale well because there are too few loop iterations for the execution pattern to be worthwhile. *adpcmc* converges on a speedup of approximately 2 when it uses 3 cores; this is limited by the dependencies between iterations and is not helped by the addition of further cores. For many of the benchmarks, the line in the energy graph is horizontal, showing that energy remains constant as more cores are used. This is because the same work is being done, but spread across more cores. The height of the line represents the overhead of the execution pattern: *bitcount_inner* has very tight loops, so the overhead is proportionally higher. It is expected that this overhead could be reduced with compiler optimisations, or by executing loop iterations in groups instead of one at a time. For *jpeg_dct* and *jpeg_huff*, energy increases because there are not enough loop iterations to overcome the overheads of filling multiple L0 caches with instructions.

The target when exploiting any form of paralellism is a perfect linear speedup, where adding extra cores provides a proportional increase in performance. This is typically difficult to achieve due to program dependencies and resource contention. For several of the applications presented, Loki achieves a linear speedup, but not perfect linear. This is due to overheads of the execution pattern:

- Extra function calls are used, which increase the code size: in order to reuse as much code as possible in these hand-modified examples, loop iterations are converted to functions which take the iteration number as an argument. The impact of this could be reduced by using cache pinning (Section 5.1.3) to keep as many instructions as possible in the



(a) Relative performance



(b) Relative energy

Figure 6.3: Performance and energy consumption as the width of the DLP group changes, relative to the baseline (non-DLP) single-core implementation. Reported figures are for benchmark kernels only.

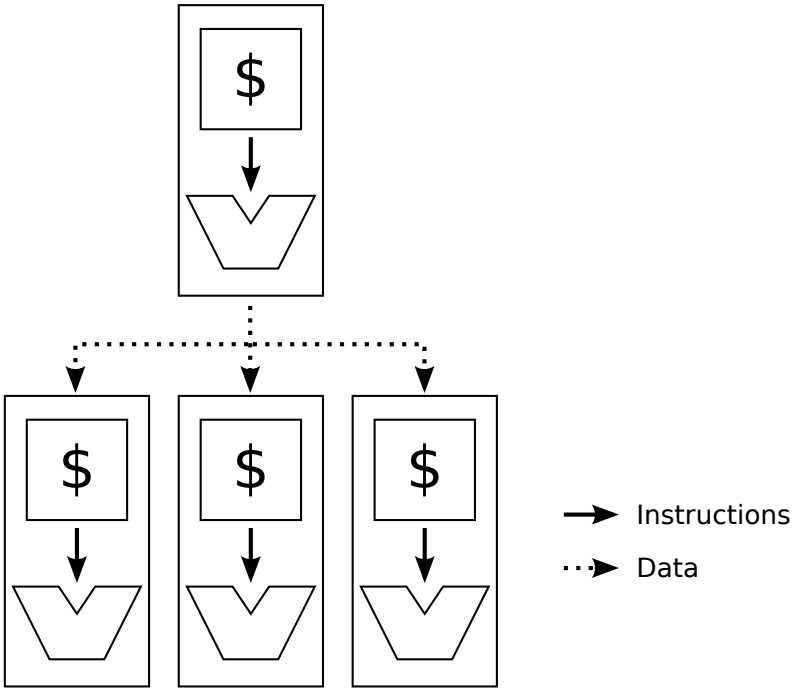


Figure 6.4: DLP execution pattern with helper core. One core is removed from the DLP group and is responsible for providing data common to all other cores.

L0 cache, or by refactoring the code to have the loop inside the function body. It is also expected that a compiler would be able to implement a lower-overhead transformation.

- In order to share data between cores, more data is made global, and must be loaded from the shared L1 cache every time it is accessed. This situation could be improved by storing read-only global values (the majority) locally in each core.

This execution pattern improves performance, but not energy. This is because much work done by Loki when in DLP mode is redundant:

- All cores have their own iteration counter, increasing register pressure, and requiring instructions on each iteration to update the counter and check whether another iteration is required.
- All cores access the same shared data held in the L1 cache, increasing contention, and slowing average response times.
- All cores fetch the same instructions and store them locally, meaning that the limited L0 cache capacity is poorly utilised.

In the following sections, I explore the use of a *helper core* to reduce redundancies concerning data (Section 6.2.3), and hardware modifications to improve instruction distribution (Section 6.2.4).

6.2.3 Helper core

In order to reduce the inefficiencies involved in multiple cores accessing and computing the same data, this section explores using one core as a *helper core* to provide common data required

by all other cores (Figure 6.4). This process is known as scalarisation and can be used on any architecture capable of simultaneously exploiting DLP and executing a sequential thread [78]. Such architectures include some vector processors which include control units and upcoming GPUs with scalar execution resources. The use of a helper core can reduce the work done by the remaining data-parallel cores and reduce contention at L1 banks, at the cost of reducing the number of cores processing the input data.

All instructions independent of live inputs to the loop iteration are considered for extraction to a helper core. This includes instructions which store constants in registers and load data from the cache. Instructions are not extracted when the cost of generating a value is the same as the cost of receiving a value from the network – this is the case when the value needs to be used multiple times and so is stored in a dedicated register. All computations used to determine whether to perform another iteration are also extracted.

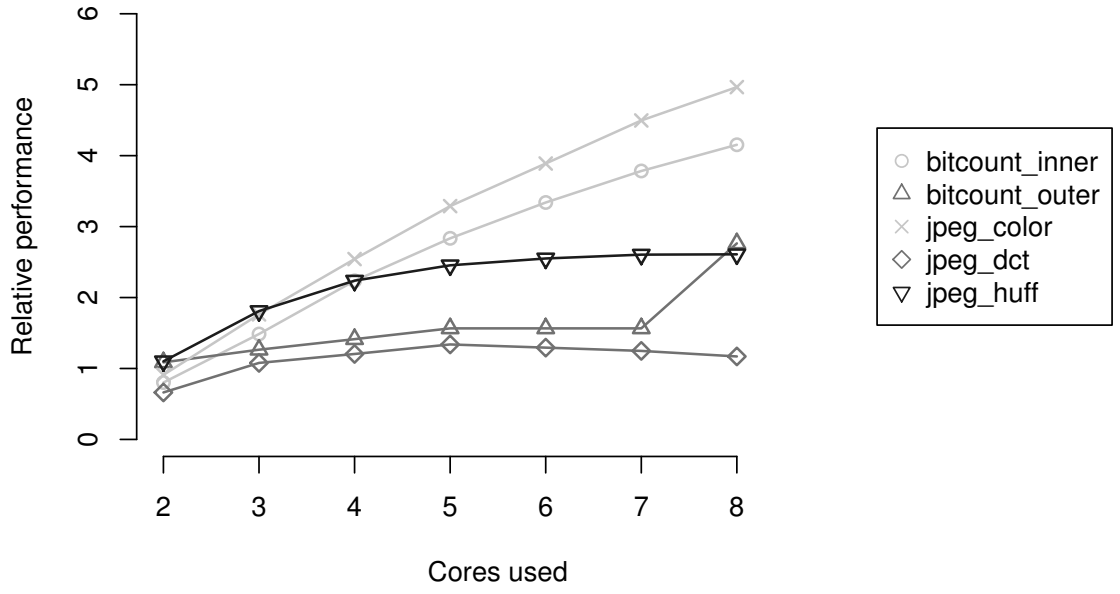
The behaviour of the benchmarks with helper cores is shown in Figure 6.5. *dijkstra* and *stringsearch* are excluded as they are too control-intensive to benefit from a helper core. *ad-pcmc* is excluded because it makes use of DOACROSS parallelism, so the cores require more decoupling than the helper core allows. In most cases, energy consumption decreases from the DLP baseline because less work is being done in total. For *bitcount_inner*, *bitcount_outer* and *jpeg_color*, total energy consumption drops as the number of cores increases because the helper core is able to provide data to more cores at once, so needs to do so fewer times. The performance impact depends on the amount of work which can be offloaded onto the helper core and the number of cores being used, and ranges from a 20% decline for *jpeg_dct* to a 16% improvement for *jpeg_color*. The performance of *bitcount_outer* suddenly rises with 8 cores because the 7 iterations of the loop now fit neatly on the 7 worker cores. Energy consumption for 8 cores is an average of 11% lower than without the helper core: data supply is 13% cheaper, and 9% fewer instructions are executed in total.

Loki's implementation of the helper core has the flexibility to deal with divergent control flow if necessary, allowing the possibility for some of the excluded benchmarks to be accelerated. Figure 6.6 shows the structure of the *dijkstra* benchmark. Usually, the helper core would only be able to help in sections which are guaranteed to be executed by all cores, so would be limited to the short initialisation phase even though the vast majority of iterations follow the same control path. Loki, however, with its multicast network and ability to dynamically reconfigure network communications, is able to do better. It is possible to generate bitmasks for the cores which take each branch, and potentially help each group separately. In the case of *dijkstra*, this would allow the helper core to continue execution into the main algorithm. The overheads of this reconfiguration are likely to be too high to benefit some applications with fine-grained control flow, but for others, such as *dijkstra*, they will be absorbed by the savings of using the helper core more of the time.

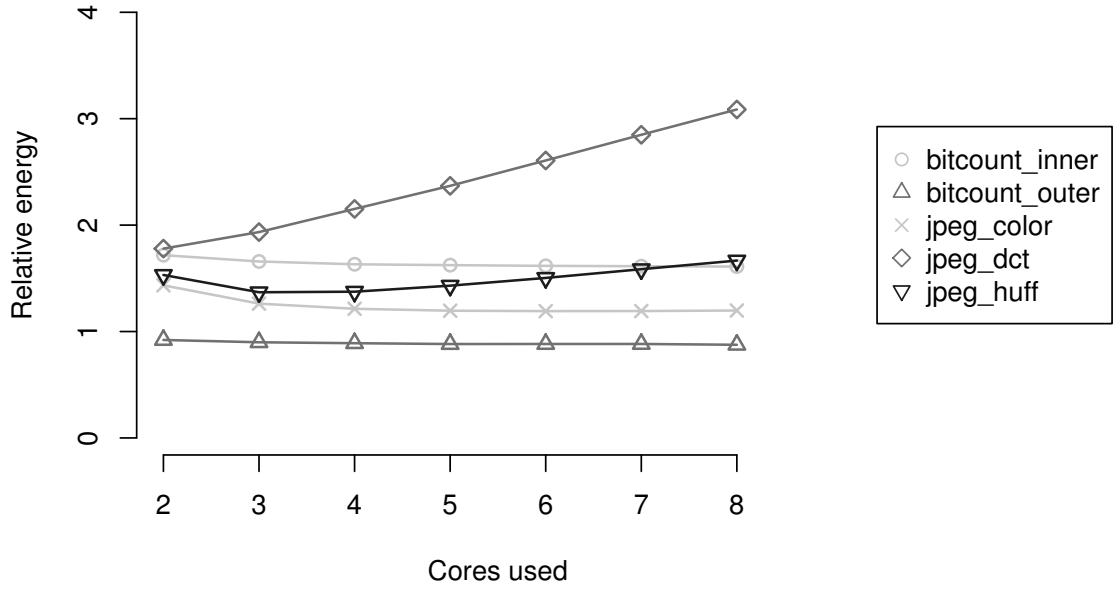
Loki's homogeneous architecture also allows the helper core to take a variety of forms: it could be a virtual processor composed of multiple cores to take advantage of further parallelism.

6.2.4 Instruction sharing

In this section, I perform a limit study on the possibility of each instruction being cached by only a single core, and distributed to all others when necessary (Figure 6.7). This optimisation is often used by dedicated SIMD architectures. It is simplest to make use of this optimisation only in cases where all cores are executing the same instructions at the same time. Instructions are distributed before being decoded: Loki's decode logic is inexpensive, and the existing core-to-



(a) Relative performance



(b) Relative energy

Figure 6.5: Performance and energy consumption as the width of the DLP group changes, when making use of a helper core.

```
for all paths {

    // Initialise

    if (startpoint == endpoint) {
        distance = 0;
        return;
    }
    else {
        // Main Dijkstra algorithm
    }

}
```

Figure 6.6: An example from *dijkstra* where divergent control flow blocks the use of a helper core.

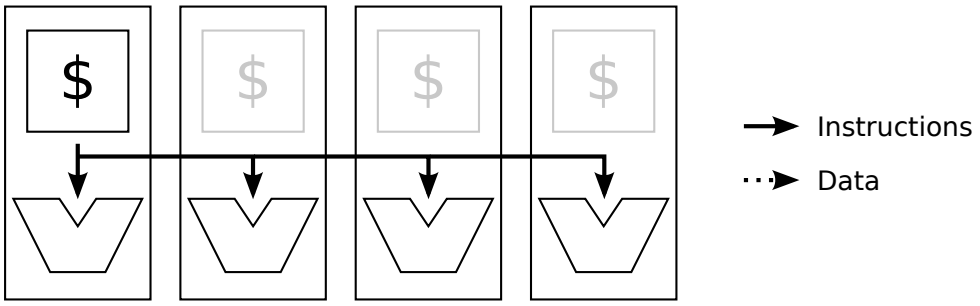


Figure 6.7: DLP execution pattern with instruction sharing. One core is responsible for issuing instructions to all others, mimicking SIMD architectures and reducing the cost of instruction supply. This core can change during execution to allow use of multiple L0 caches.

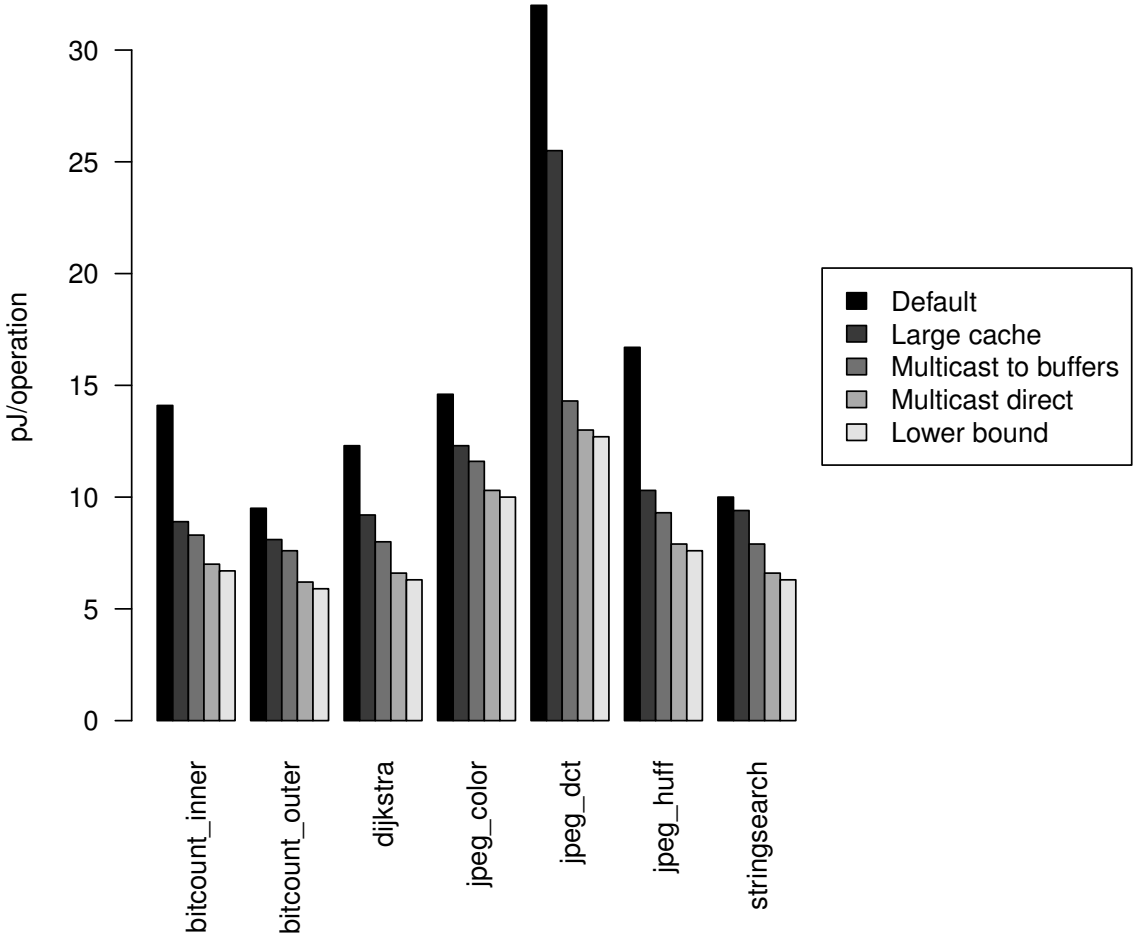


Figure 6.8: Total energy consumption of various instruction sharing strategies. Results are for 8 cores. *Default*: DLP with no instruction sharing; *Large cache*: each core has 8× the L0 cache capacity; *Multicast to buffers*: instruction is read from one L0 cache and distributed to instruction buffer of all other cores; *Multicast direct*: instruction is read from one L0 cache and distributed directly to decode stage of all other cores; *Lower bound*: instruction distribution is instant and consumes no energy.

core buses can be used, rather than requiring an additional wider bus for decoded instructions. When sending instructions on these buses, there is no contention with other data sent between cores because SIMD execution requires there to be no dependences between loop iterations, so no data needs to be transferred. Figure 6.8 shows the energy consumption of a range of benchmark kernels for various instruction sharing mechanisms. In the limit case (*Lower bound*), this will cut instruction supply costs (including L1 accesses and network activity) by the number of cores. With no duplicate instructions in the cores' L0 caches, the L0 cache capacity of the group scales up by the number of cores. Access costs remain constant, however, since only a single cache is accessed at a time. Techniques for switching between different cores' instruction caches have been demonstrated previously by the Elm architecture [14].

A number of intermediate configurations are also presented to show where the savings come from and demonstrate the behaviour of more-realistic implementations:

- *Larger cache* shows the energy impact of increasing each core's L0 cache capacity by 8 times without affecting access costs. (This is effectively what happens when 8 cores

share their instruction caches.) The extra cache capacity also improves performance over the *Default* case by an average of 14%.

- *Multicast to buffers* shows the savings when only one core's cache needs to be read. The instruction is sent on the existing multicast network to the instruction buffers of all other cores.

- *Multicast direct* shows the savings when instructions are sent directly to the pipeline registers of other cores, bypassing the instruction buffer. A larger multiplexer in the fetch pipeline stage would be required to accommodate this extra instruction source; the costs of this are ignored.

This technique is only suitable for DOALL parallelism, since the cores all execute the same instruction at roughly the same time (there may be some decoupling if instructions are delivered to each core's instruction buffer). Any significant control flow will also make the optimisation less effective, as fewer cores will be executing the same instructions simultaneously, though Loki's multicast crossbar may make it possible to have multiple independent SIMD groups executing simultaneously. It is assumed that it is possible for data to be arranged in memory such that the effects of additional contention at the L1 banks are negligible.

Multicast to buffers is expected to be the most realistic implementation of instruction sharing, as it requires minimal modifications to the hardware, and in most cases, there is only a small additional improvement for the more drastic optimisations. Instruction supply energy is reduced by an average of three quarters over the default 8-way DLP implementation (corresponding to a 35% drop in total energy consumption) and performance improves by 14% due to the increased cache capacity.

In some cases, it may be desirable to distribute instructions which have already been decoded. This was not explored here because Loki's decoders are relatively inexpensive compared to the extra information which would need to be communicated, and because additional buffering in the middle of the pipeline would be required. Distributing encoded instructions allows existing buffering and communication networks to be used.

6.2.5 Worker farm

A worker farm allows load balancing between cores, and is used when there are many independent tasks to be performed, with a high variance in execution time. Instead of being allocated a static subset of tasks, worker cores request new tasks from a master core when necessary. This allows cores which complete their work quicker to continue being productive, while slower cores do not hold up the others.

Worker farms have very similar aims to task stealing, where cores are assigned tasks, but can *steal* from other cores if they finish early. It has been shown that there is no best implementation of task stealing for all applications, and that flexibility is required [113] – this is possible with the software implementation that Loki provides. Low-overhead messaging is found to be sufficient for many schedulers to operate efficiently. The worker farm was chosen for this work because it is simple to implement in software, and is not often used on general-purpose homogeneous architectures.

A worker farm, as implemented on Loki, is illustrated in Figure 6.9. One core in the tile is reserved as the master and keeps track of which tasks have been allocated to workers, and which tasks have been completed. Workers request new tasks by sending a unique identifier to the master. The master waits for a new identifier (blocking until one arrives), and sends out the

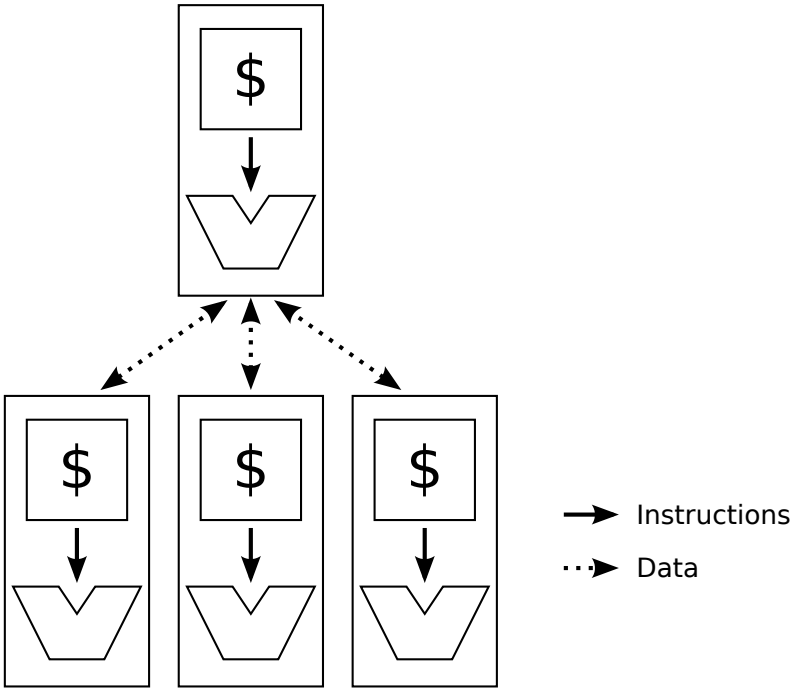


Figure 6.9: Worker farm execution pattern. One core is designated the *master* and is removed from the DLP group and made responsible for distributing tasks among the other cores. Workers let the master know when they would like to be issued more work.

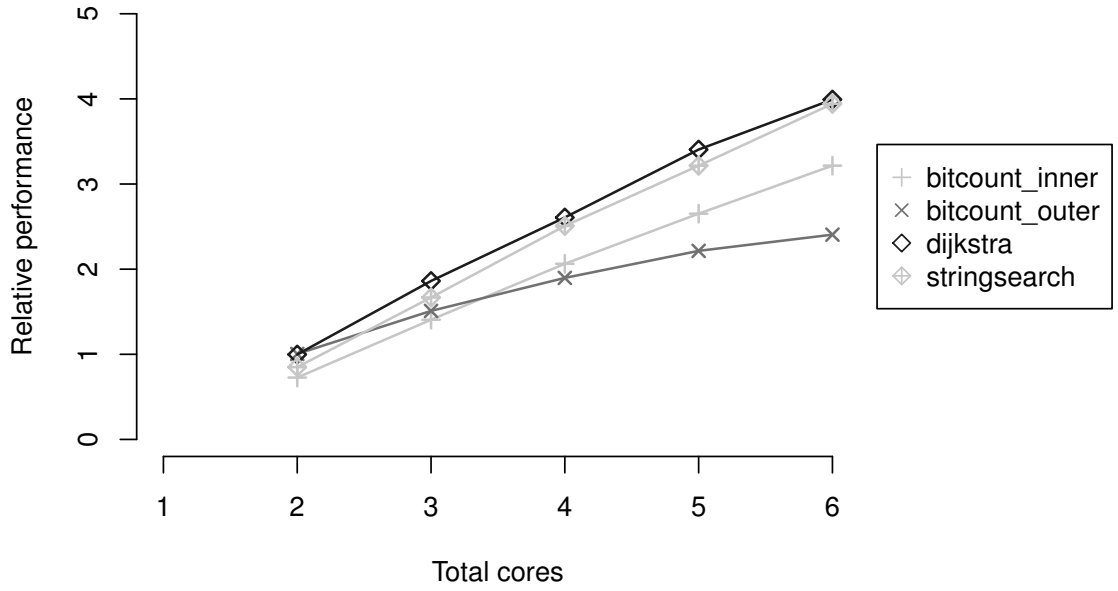
next task to be performed. The implementation is currently limited to 5 workers because the master core has only six data input channels, one of which is reserved for data from memory, and Loki requires that each channel has only one writer to simplify arbitration. Steps towards removing this limitation are discussed later.

Loki's worker farm implementation therefore offers a tradeoff: DLP offers higher potential throughput by using more cores, but is vulnerable to variance in task length; worker farms spend a core to offer better load balancing. This is a similar tradeoff to that of scalarisation, where one core is removed from the group to fetch data and perform computations common to the rest of the cores.

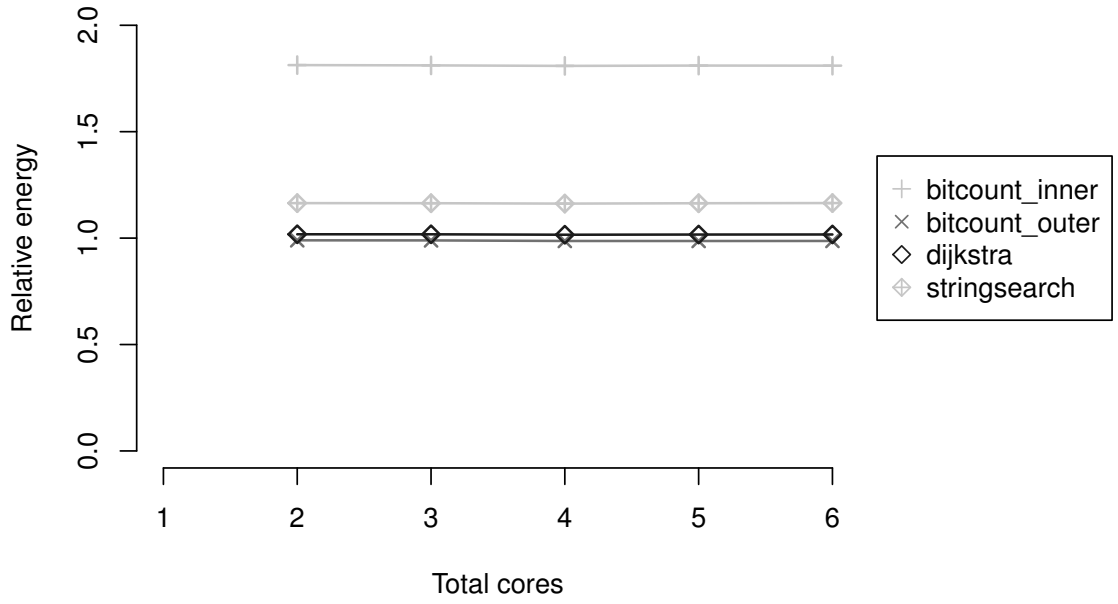
Energy for a worker farm is always going to be higher than for normal DLP - all of the same work is being performed, plus the overheads of the master core issuing tasks. This technique is targeted at loops with variable iteration execution times, and may be able to improve performance in these cases.

Figure 6.10 shows how performance and energy consumption vary as the number of cores in the worker farm changes, and Figure 6.11 shows how the largest worker farm (5 workers and 1 master) compares with a DLP group of the same size, and with the largest DLP group tested. Kernels which contain no control flow are excluded as these will have very little variance in execution time, and so will not be helped by the worker farm execution pattern.

bitcount_inner and *stringsearch* show no improvement in performance when moving from DLP to a worker farm. Their loops are tight, so the overheads of communicating with the master core are not worthwhile. Performance losses are less than the $\frac{1}{6}$ which might be expected by removing one of the cores from the data-parallel computation, indicating that load balancing is helping slightly. *bitcount_outer* makes more effective use of its six cores than the plain DLP implementation, but is not able to match the total throughput of the 8 core version. *dijkstra* shows a large performance improvement of 30% over 6-core DLP, and also outperforms the 8 core im-



(a) Relative performance



(b) Relative energy

Figure 6.10: Behaviour of worker farm execution pattern relative to baseline single-core benchmark kernels.

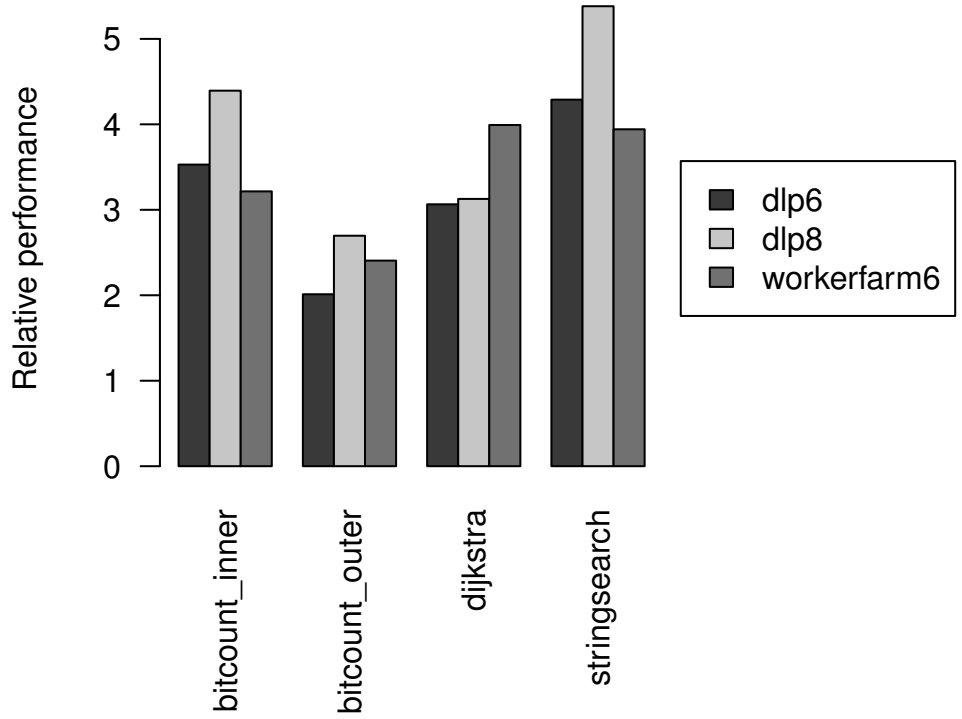


Figure 6.11: Performance comparison between worker farm and DLP for a selection of benchmark kernels. All figures are relative to the single core base case.

plementation comfortably. This indicates that there is a high variability in the execution times of *dijkstra*'s loop iterations, and that in some cases, intelligent management of computation can result in lower execution times, even when far less execution resources are available.

The number of worker cores is limited to five in the current implementation, since the master core has only six network inputs, one of which is reserved for memory communications, and each input can have only a single writer to eliminate the need for arbitration. There are a number of ways in which it would be possible to go beyond the limit of 5 worker cores. Exploring these is left for future work.

- Allow multiple writers to each network channel by using an additional arbiter at each buffer.

- Multiplex multiple worker cores onto a single channel at the master core using synchronisation in software.

- Provide multiple master cores, perhaps in a hierarchy. Having a hierarchy of master cores would allow an execution pattern very similar to MapReduce [35]: each master receives a block of work to do and splits it up among its workers, before combining all of the partial results and reporting back.

- Treat network buffers as a cache of available communication channels, as with the Tilera architecture [132]. When data arrives from a new source, swap out the contents of an existing buffer to make space.

6.2.6 Parallelism extraction

Extracting data-level parallelism automatically is usually harder than expected, as the compiler is often unable to prove that there are no loop-carried dependencies. Automated extraction techniques often rely on speculation to overcome the difficulty of proving independence of loop iterations [136]. There has been much work in this area, but there is still a long way to go before automatic techniques are able to identify all of the available data-level parallelism.

Many parallel programming languages provide some form of a *parallel_for* construct, allowing the programmer to explicitly tell the compiler that the desired transformations are safe [1, 95].

HELIX [29] is a technique used to exploit DOACROSS parallelism where further code transformations are made to minimise the impact of the required communication and synchronisation between iterations. HELIX is also predictable enough that it is possible to compute whether or not the transformations are effective, allowing for fully-automated parallelisation of sequential code. It is found that cheap communication is very useful in exploiting this form of parallelism, so I expect it to map well to Loki.

Lee et al. demonstrate a technique for automatically extracting the scalar part of a data-parallel program to reduce redundant work performed, which can then be executed on a helper core [78]. They observe that "temporal-SIMT is a particularly good match for scalarization"; temporal-SIMT is a class of data-parallel architectures where each pipeline is capable of independently fetching and executing instructions. Loki can therefore be seen as belonging to this class.

The decision of whether to use a worker farm could be made be estimating the variance of execution times for individual loop iterations and the number of iterations to be performed on each core. The highest expected execution time for a single core in a group of n cores can then be computed, and if this is greater than the time taken by $n - 1$ cores to execute a balanced workload, then a worker farm is likely to be the better choice.

6.2.7 Conclusion

It is possible to greatly increase the performance of data-parallel kernels using DLP whilst simultaneously reducing energy consumption. Improvements can be seen even when there are loop-carried dependencies.

A simple DLP implementation, where a number of cores independently execute the same code with different data, is able to achieve an average $3\times$ speeedup on 8 cores with less than 50% extra energy consumed. It is expected that both of these metrics can be improved further through the use of compiler optimisations to reduce the total number of instructions executed.

It would be feasible to extend the DLP group across multiple tiles. This would require an extra initialisation phase, where a master core delegates work to the master core of each tile. Multicast is not possible across tiles, so messages would have to be sent repeatedly, but the benefits of having more cores working on the problem would often outweigh these costs. Future work will identify whether additional interconnect between tiles can aid such execution patterns.

By default, there is often lots of repeated work involved with this execution pattern: all of the cores need to fetch the same instructions and compute some of the same values. Using a helper core can reduce redundancy in data supply, reducing energy consumption by an average of 11% and improving performance by up to 16%. Sharing instructions between cores can greatly improve the efficiency of instruction supply, providing an average energy reduction of

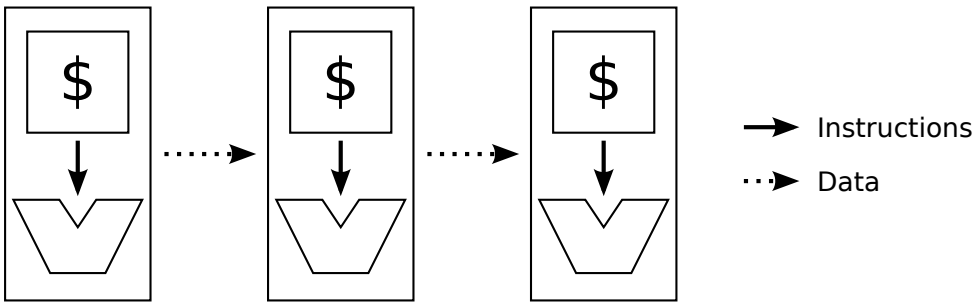


Figure 6.12: Task-level pipeline execution pattern. Each core performs an independent processing task on the received data, and sends the result on to the next core.

35% and performance improvement of 14%. These two optimisations are orthogonal and can be applied in combination to further improve execution characteristics.

In some cases, the worker farm execution pattern offers an improvement over plain DLP because of better load balancing across the cores. It is therefore another viable option available to the programmer or compiler. This pattern has the potential to work well on Loki because of its low latency communication, but is limited by the restriction of only allowing a single writer to each network channel.

All of these modifications to the base DLP pattern were beneficial in some cases, but not in others. Furthermore, there was no modification which was out-performed by the others in all cases. The flexibility to choose an execution pattern and then modify it to suit the application allows for improved performance and energy efficiency.

6.3 Task-level pipelines

The task-level pipeline execution pattern involves each core doing an independent chunk of work on the input data, before passing the result onto the next core (Figure 6.12). The concept can be extended so that each pipeline stage consists of a virtual processor, which may itself be made up of a number of cores. The aim is to improve locality by having each core (or virtual processor) working on its own part of the program, whilst also taking advantage of parallelism in the program by executing multiple pipeline stages simultaneously. This technique is useful for some loops which have cross-iteration dependencies, as the length of the loop bodies is decreased, and so throughput is increased. The result is that a large loop is transformed into multiple smaller decoupled loops which can execute in parallel.

This technique has been successfully implemented on traditional multicore architectures, but there is more flexibility on Loki: there are many more cores available; it is easier to balance the load between pipeline stages by using parallelism within each stage; and faster and cheaper communication allows for finer-grained stages. I also explore the use of pipelining for reasons other than improving performance: energy consumption can be reduced by making use of the increased cache and register capacity of multiple cores.

6.3.1 Evaluation

JPEG compression and decompression are usually considered ideal examples for implementing task-level pipelines: the image is split into small blocks of pixels, and each block passes through several independent stages of processing. The implementation in MiBench, however, has been

heavily optimised to reduce the memory footprint, and as a result is very complex. The scale of the modifications required to extract pipeline-level parallelism were considered too large to be able to perform a worthwhile comparison afterwards. I instead make use of a number of simpler benchmarks, which allow rapid exploration of different implementations.

In order to minimise changes made to the existing code, wrapper functions were placed around existing functions to send and receive data in a pipeline fashion. The wrapper functions performed any initialisation, such as setting up a network connection to the next core in the pipeline; repeatedly executed each core's task, whilst receiving arguments and sending results over the network; and tidied up after the loop, for example by sending a final result back to the main thread. This approach is likely to have higher overheads than one which the compiler is aware of, so is expected to underestimate any benefits of pipeline-level parallelism.

Again, since only a subset of application kernels in the benchmark suite contain pipeline-level parallelism, case studies are performed on the individual programs. Pipeline parallelism is expected to be more abundant in larger programs.

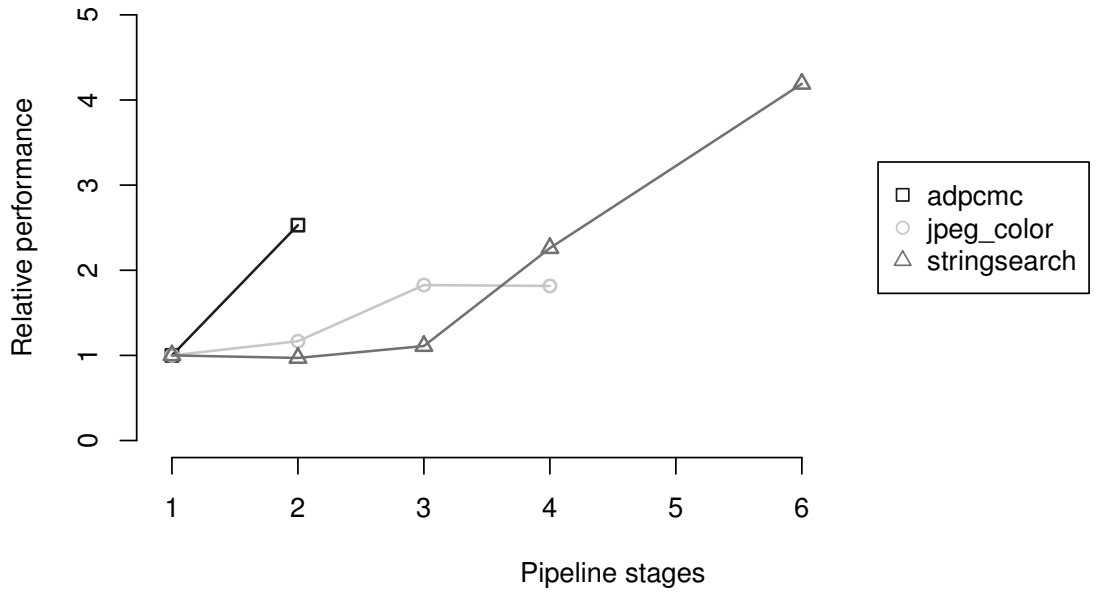
The color space transformation phase of *jpeg* was extracted as a loop which could make use of pipeline-level parallelism. Each iteration consists of loading RGB values from memory, and then computing YCbCr values from them. Loading of all input values and computation of each output value were treated as separate tasks: four in all. These were then distributed across 2-4 cores.

The *stringsearch* benchmark consists of two main phases: initialising a 256-entry table, and then traversing the string. These tasks were placed on separate cores to form a two-stage pipeline, and the table initialisation phase was split into up to five separate sub-phases to extend the pipeline further.

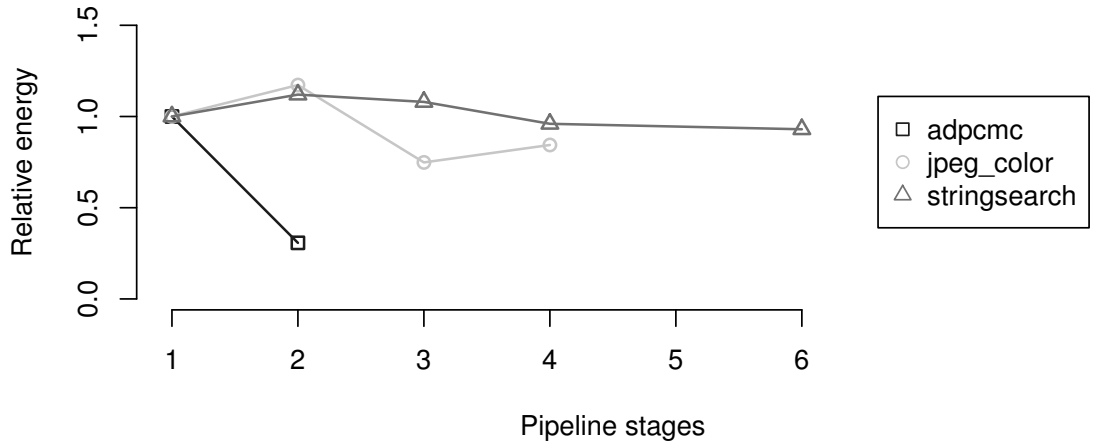
adpcmc was mapped to a pipeline structure slightly differently. It does not possess traditional pipeline-level parallelism as there is a loop-carried dependency between the end of one iteration and the beginning of the next. Instead, the loop body was mapped to a pipeline-like structure which had an additional network connection from the final stage to the first stage. The instruction sequence was split naively at basic block boundaries such that each pipeline stage could fit entirely within a single core's L0 cache, and variables which were live across this split were communicated as soon as their values were generated. A small amount of rescheduling was required to ensure that values were sent and received in the same order. This transformation effectively creates a virtual processor which is tailored to the application by providing sufficient instruction cache space, and moves the computation during execution to a location where instructions can be accessed cheaply. This is in contrast to typical execution patterns, where instructions are fetched to the core which holds the relevant data. Finally, the standard hand-optimisations described in Section 4.3.1 were applied – with more registers available across the pair of cores, more data could be stored locally. These optimisations resulted in a further 15% reduction in static code size over the assembly-optimised single-core baseline. An additional optimisation applied, which is expected to be within reach of a standard optimising compiler, was to attempt to send data along the pipeline as early as possible, and receive data as late as possible, to increase the duration for which the cores can execute concurrently.

Figure 6.13 shows how performance and energy consumption change as the number of cores in the pipelines increase. The execution pattern adds negligible overhead in these cases, and all implementations start roughly at parity with the single core baseline.

Energy consumption for *jpeg_color* improves by 25% with three cores due to the improved cache performance when each core's task is restricted. Moving to four cores does not improve



(a) Relative performance



(b) Relative energy

Figure 6.13: Relative energy and performance of task-level pipelining for a selection of MiBench kernels. All figures are relative to a single-core baseline implementation.

cache behaviour any further, but does increase the number of operations performed, so energy consumption increases. Performance also improves by 80%.

stringsearch does not improve much when using short pipelines because the tasks done by each stage are so imbalanced. As the pipeline grows, performance rapidly improves, ending at 4.2× with six cores. Energy consumption reduces slightly due to improved caching.

adpcmc sees a 70% reduction in energy and a superlinear speedup of 2.5× on two cores. These improvements have three sources: greatly improved cache behaviour, with 100% of instructions now being stored in L0 caches; more aggressive optimisations made possible by the extra registers available; and concurrent execution on the two cores. 1.56 instructions were executed per cycle, indicating that a large amount of instruction-level parallelism was unlocked by the transformation. It may be the case that further improvements can be made with a more careful mapping of instructions to cores. This pipelined implementation outperforms execution on a single core with twice the L0 cache capacity by 2.0× and reduces energy by 20%.

This technique of using pipelining to increase the available resources has similarities with cache pinning as described in Section 5.1.3. Both transformations involve making maximum use of a core's small L0 instruction cache, since it is much cheaper to access than an L1 memory bank. When cache pinning reaches the end of its locally stored instructions, it moves the required instructions from the L1 memory to a local structure. In contrast, when pipelining reaches the end of its locally stored instructions, it moves the required data to another core, where the instructions are already cached. This question of whether instructions should be sent to where the data is stored, or whether data should be sent to where the instructions are stored becomes increasingly relevant on fine-grained architectures such as Loki. Pipelining is more scalable – the length of the pipeline can be extended almost arbitrarily in order to improve caching behaviour, whereas cache pinning is only useful for codes which are only slightly too large to fit in the cache.

The technique also bears similarities to inter-core prefetching [62]. Inter-core prefetching allows multiple cores to cooperatively execute a single thread by having a compute thread and one or more prefetch threads. The compute thread migrates between cores, following the prefetch threads, and finds the data and instructions it needs waiting in the local cache. Loki's implementation is much more fine-grained and light-weight: each slice of the program is very small, and instead of a complete context switch taking thousands of clock cycles, only small number of live variables are sent, with near-zero overhead. The pipeline-like approach also allows instruction-level parallelism to be exploited by executing multiple pipeline stages simultaneously.

6.3.2 Parallelism extraction

Decoupled software pipelining (DSWP) [99] provides a fully-automatic mechanism for extracting pipeline-level parallelism from programs. It breaks loop bodies into multiple smaller sections such that data dependencies between loop iterations never cross section boundaries, and so each section can be executed by a different thread. This helps to decouple the threads, preventing variable execution times from slowing down the whole pipeline. There is still some degree of coupling, determined by the size of the communication buffers – a core will stall if it is unable to write any more data to the buffer. DSWP claims wider applicability than DOALL parallelism, and better performance than DOACROSS. The technique would work well on Loki and could be combined with the optional heuristic that each pipeline stage should fit in an instruction packet cache to reduce energy.

Huang et al subsequently found that DSWP can be used as an enabling transformation for other forms of parallelism [55]. For example, some of the loop sections contain no dependencies, allowing DOALL parallelism to be extracted, and other loop sections may contain only occasional dependencies, making speculation useful. This extra parallelism can be used to balance the pipeline stages to reduce bottlenecks and improve throughput.

In cases where it is not straightforward to automatically extract pipeline-level parallelism, it would be possible for the programmer to provide hints to the compiler, for example using SoC-C's `pipeline` construct [108].

Linear logic [5] has been shown to be a useful tool for code exhibiting pipeline parallelism, with information being passed from core to core and exactly one core needing access at any one time. This restriction makes memory management simpler and can allow the optimisation of replacing memory copies with reference passing.

6.3.3 Conclusion

Pipeline-level parallelism is another possible way to improve performance and reduce power consumption on tightly-coupled cores.

Performance can be improved by executing different parts of different loop iterations simultaneously – this can allow parallelism to be exploited even when there are dependencies between iterations. Energy can be reduced by having less code on each core, thereby improving caching behaviour. Both performance and energy consumption can be improved by eliminating function call boundaries and sending data directly between cores. This approach may also unlock some instruction-level parallelism, as the remote core may be able to begin execution before it has all of the function arguments. Improvements are limited by the ability to balance load across multiple cores. Loki's inexpensive communication mechanisms allow even tightly-coupled regions of code within a single pipeline stage to be split across multiple cores to improve load balance.

Specialising the task done by each core generally means that its cache performs better. Furthermore, techniques such as cache pinning become viable more often, which are able to improve performance and energy consumption further.

Spreading a program across multiple cores effectively increases the sizes of important structures such as caches and register files, without increasing the energy required to access them. This allows parallelism to be exploited and more-aggressive code optimisations to be used, whilst also reducing the total energy cost of a program. Scaling is not perfect, as some information usually needs to be replicated.

This ability to increase the effective sizes of fundamental structures with relatively low overhead suggests that it may be worthwhile deliberately underprovisioning individual cores, with the expectation that an appropriate number will be grouped together to execute the task at hand. This would mean smaller and lower-power building blocks for virtual architectures, and allow resources to be allocated at a finer granularity.

6.4 Dataflow

Dataflow is an execution paradigm where the change in the value of a variable automatically forces recomputation of any variables which depend on it. Dataflow execution graphs are similar to software pipelines, except that they are usually finer-grained. Given enough computation resources, dataflow is capable of computing a result in the minimum possible time, as all operations on the critical path will execute sequentially as soon as their inputs are available. Typically,

it is possible to execute tens to hundreds of operations at once, but the challenge is in identifying these independent operations and mapping them to the available hardware resources quickly enough [83].

Architectures designed to execute in a dataflow manner can make use of *static* or *dynamic* dataflow. With static dataflow, each communication link can have at most one item on it. When all operands for a function are ready, they are consumed, and the source components are notified that the channels are available once more. Dynamic dataflow allows channels to contain multiple operands, each associated with a different colour or tag. Functional units wait until a pair of operands of the same colour are available before consuming them. Colours allow concurrent execution of multiple copies of the same code, but the matching process can be expensive.

Coarse-grained reconfigurable arrays (CGRAs), for example, typically use a static schedule and a predictable mesh network to communicate between functional units [100]. Some CGRAs allow multiple operations to be time-multiplexed onto each functional unit to allow larger dataflow graphs to be accelerated.

Many modern superscalar processors provide a limited form of dataflow using out-of-order execution: instructions are marked as ready to execute as soon as their operands have been computed. Combined with speculation, the technique allows a steady flow of instructions to be supplied to the functional units; without speculation, it is more difficult to find enough independent instructions to make use of the available resources. The extent to which dataflow can be exploited is limited by the size of the issue window and number of functional units, and lots of extra logic is required to keep it all working (register renaming, reorder buffer, etc.).

The TRIPS architecture makes use of explicit data graph execution (EDGE) to exploit parallelism inherent in dataflow graphs [28]. Instead of specifying registers as locations for results to be stored, EDGE operations name the instructions which directly make use of the result. Instructions are then statically grouped into hyperblocks, which are dynamically scheduled by the processor. By minimising communication between hyperblocks, resources local to the hyperblock can be used more often, which are cheaper to access than global structures.

Dataflow can be implemented on Loki by placing a single instruction on each core, with all inputs received from the network, and all outputs sent onto the network (Figure 6.14). This is made easier by some Loki-specific features: persistent instruction packets (Section 3.2.1) allow endless re-execution of a section of code without the need for branches or jumps, and Loki's blocking channel communication mechanism means a core will wait for all of its inputs to arrive before executing an instruction, and will stall if there is no network buffer space to send the result. Loki's implementation lies somewhere between static and dynamic dataflow: network buffers mean that multiple operands can be on a single channel at any time, providing decoupling between functional units, but the lack of tag-matching logic means that if there are multiple operands to be used by a single function invocation, they must all arrive in the same order.

Loki's implementation of dataflow has the potential to scale to hundreds of functional units without incurring the overhead of complex hardware, such as that found in superscalar processors. If the overheads of pure dataflow are ever too high, it is possible to place multiple instructions on a single core, and fetch them from the cheap local storage, as CGRAs do. Instructions could be grouped using techniques similar to the TRIPS architecture, described above, to use local resources as frequently as possible and minimise communication between cores. Loki's flexible communication network provides benefits over CGRAs, which often can only communicate with their nearest neighbours, sometimes requiring functional units to be spent on forwarding data to the required location, rather than performing any useful computation. In

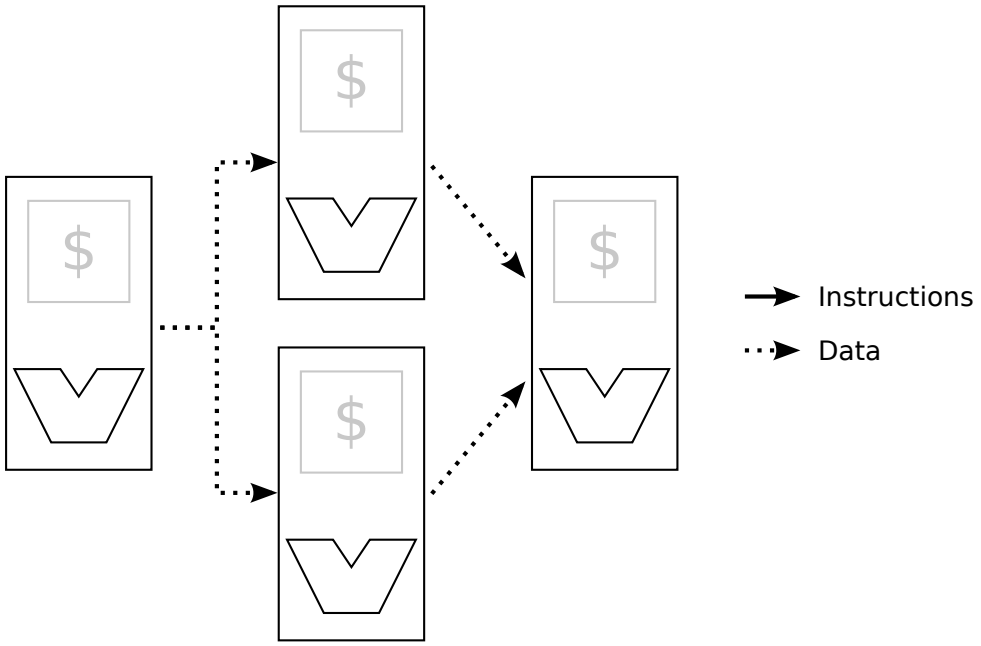


Figure 6.14: Dataflow execution pattern. Each core is capable of receiving up to two operands from the network, and sending its result to any subset of cores on the same tile. When executing only a single instruction on a core, it does not need to be issued repeatedly, and the entire front end of the pipeline can be power gated.

contrast, any Loki core can communicate with any subset of the eight cores in its tile, whilst also being able to communicate off-tile if necessary. The cost of this flexibility is increased interconnect energy consumption.

When the task given to each core is so restricted, switching activity within the pipeline is reduced, and it may be possible to deactivate unused components to save energy. The cost of dataflow execution is greatly increased use of the network, so this must be compared with the expected gains before deciding whether to use the dataflow technique. A disadvantage of Loki's approach compared to a specialised architecture is that although much of the pipeline is inactive and does not consume much energy, the extra components mean that the functional units are further apart, and so communication between them is more expensive.

6.4.1 Power gating

One of the advantages of the dataflow execution pattern is that it reduces switching activity in each pipeline. In this section, I explore completely deactivating unneeded parts of the pipeline in the extreme case of one instruction on each core.

If the instruction being executed does not change, the following parts of the pipeline can be power gated after the first access:

- Fetch pipeline stage
- Pipeline register between fetch and decode pipeline stages
- Decoder
- Channel map table

- The register file is often not needed. An inexpensive comparison is required to see if the instruction has a destination register (5 bit comparison with zero). If the instruction reads from the register file but does not write to it, this only needs to be performed once, as the data will never change.

- It may be possible to power gate sections of the pipeline register between decode and execute stages: the operation, channel, and some operands are always the same.

- Depending on the low-level implementation of the ALU, unused functional units can be power gated.

Furthermore, the energy model for the ALU suggests that when the operation being performed doesn't change, the energy consumption drops by an average of approximately 0.4pJ per operation (Table 4.4).

Although much of the pipeline is unneeded when one instruction is executed repeatedly, several new components experience an increased workload. There is a large increase in the activity of many of the network components: buffers, arbiters and the relatively long wires of the tile network. Dataflow execution is only beneficial if these costs are outweighed by the savings in reduced pipeline activity.

For the rest of this section, I say that a core is in *dataflow mode* if it is executing a single persistent instruction, and is able to bypass much of the pipeline as described above.

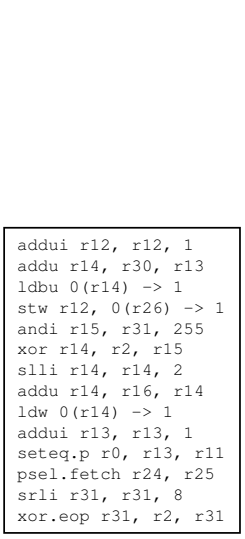
Techniques such as Δ-dataflow exist which reduce switching activity even further [84]. If a value does not change after recomputation, a single out-of-band "no change" bit is sent through the network in its place. This helps prevent oscillations in situations where the same physical wires are shared by multiple separate channels, and allows the data arrays of network buffers to be bypassed in some cases. Application of Δ-dataflow to the Loki architecture is left for future work.

6.4.2 Case studies

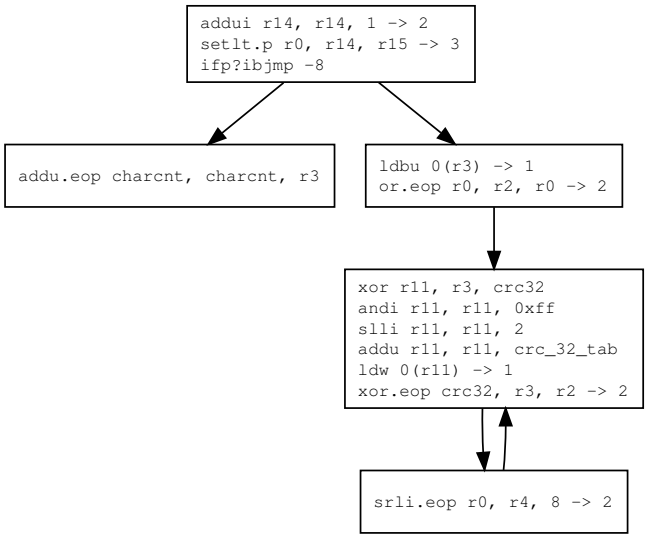
Since my exploration of dataflow is currently limited to a single tile of eight cores, only very small kernels can be used. There are not many of these which run for a significant length of time in MiBench, so I relax the dataflow concept slightly to allow a small number of instructions on each core, and I perform case studies on the main loop of *crc* and one of the *bitcount* implementations to determine how best to map them to the Loki tile. Larger dataflow graphs would need to be arranged to take into account the greatly reduced bandwidth between tiles, but due to the Rentian locality observed in software [43], this is not expected to be a major limitation. Additional tile-to-tile interconnect could be added to alleviate any bottlenecks; this is discussed later.

Inline assembly code is added to the benchmarks because the compiler is not yet able to produce efficient communication code – there is no reason why it would not be able to do this in the future. A small amount of preparation code is required before the loop begins to set up the required network connections and to send the values needed for the first iteration.

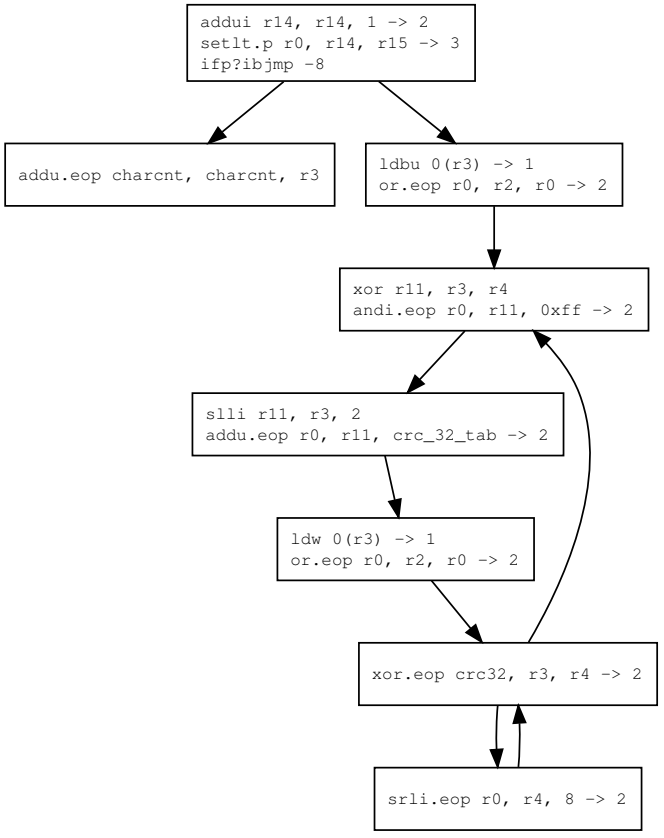
The baseline is a single Loki core running an assembly-optimised benchmark – the same code is used in the dataflow implementation, but spread over multiple cores.



(a) One core

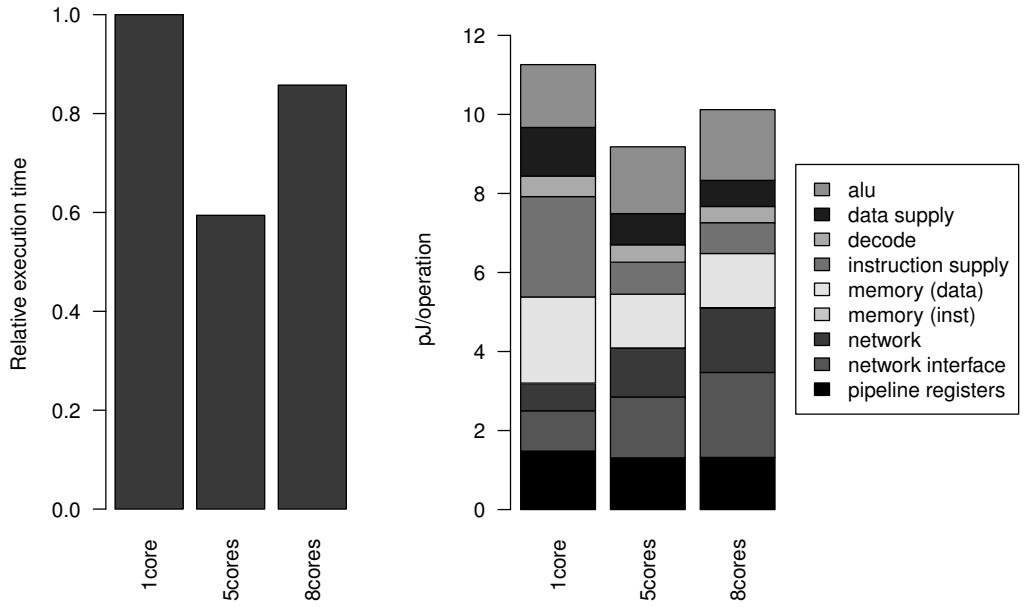


(b) Five cores



(c) Eight cores

Figure 6.15: Dataflow mappings of the *crc* kernel to multiple cores. Register indices between 2 and 7 correspond to network buffers.



(a) Relative performance **(b)** Energy distribution

Figure 6.16: CRC dataflow behaviour. Instruction supply is more efficient when using the dataflow execution pattern, at a cost of increased network (buses and arbiters) and network interface (channel map table and network buffers) activity.

CRC

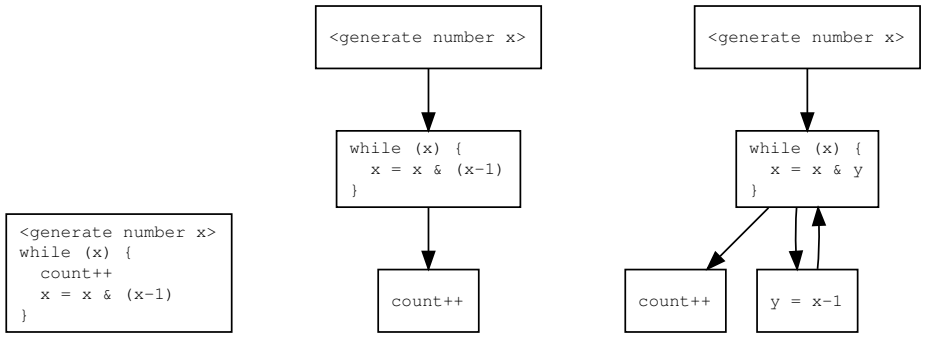
Two different dataflow mappings for *crc* were explored: one which emulated traditional dataflow, with as few instructions on each core as possible (Figure 6.15c); and one where the critical path was all placed on a single core to eliminate network latency and improve performance (Figure 6.15b). All cores except the first one are able to execute their instructions out of the cheaper instruction buffer; the first core is also responsible for executing the outer loop of the benchmark, which does not fit in the buffer. Cache (or buffer) pinning could be used to improve instruction supply energy in this case.

Figure 6.16 shows the energy distribution for the single core baseline and the two dataflow mappings. It can be seen that the dataflow mappings reduce the cost of instruction supply (and, to a lesser extent, decoding and pipeline registers), at the cost of increased network activity. The 8 core mapping increases network costs much more than the 5 core implementation, but offers only a small reduction in pipeline activity, so is not as energy efficient. The 5 core case performs 68% better than the baseline and consumes 19% less energy, while the 8 core case is only 17% faster and consumes 11% less energy. This shows that the additional latency incurred from network communication on the critical path can have a large impact on performance, and that the overheads of the additional communication can outweigh the advantages of being able to put more cores in the low-power dataflow mode.

Bit count

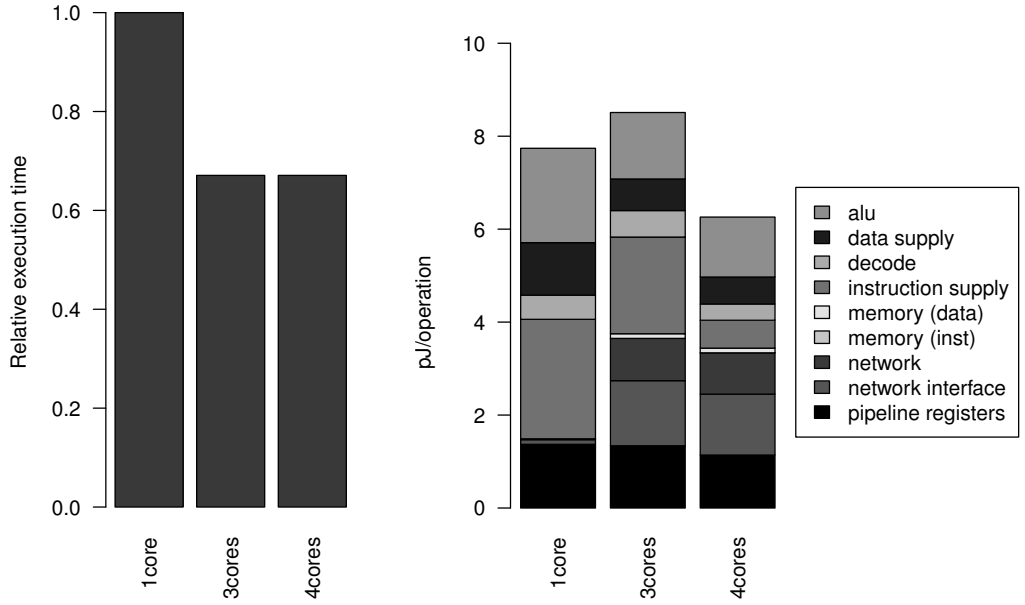
One of *bitcount*'s counting methods with a particularly tight loop was selected for these experiments.

Again, two dataflow mappings were explored: one which put the critical path on one core (Figure 6.17b), and one which spread the instructions across a larger number of cores (Figure 6.17c).



(a) One core **(b)** Three cores **(c)** Four cores

Figure 6.17: Dataflow mappings of a *bitcount* kernel to multiple cores.



(a) Relative performance **(b)** Energy distribution

Figure 6.18: Bit count dataflow behaviour.

It would be possible to extend the dataflow network to a fifth core by separating the computation of x from the branch condition. This option was not explored, however, as the overheads of doing so were considered prohibitive. This transformation would extend the critical path, as additional network communication would be required. A more subtle problem is that the computation of x becomes completely decoupled from the decision of whether the next value of x needs to be computed. This allows x to be computed many iterations in advance, which can often be beneficial, but it means that the network and buffers must be drained of surplus values at the end of each iteration of the outer loop. This requires dropping out of the persistent dataflow packet and executing extra instructions on each core. With an inner loop as tight as the one in the bit count benchmark, this was not considered worthwhile.

Unlike *crc*, *bitcount* observes a significant energy improvement when more cores are used: energy increases by 9% over the baseline when three cores are used, but decreases by 18% when four cores are used. This is because the kernel has very few instructions, so even being able to put one extra instruction on a core in dataflow mode can make a big difference. Performance for the two dataflow mappings is almost identical, at 50% faster than the baseline. Performance

does not decrease when the critical path is split across multiple cores because some of the network communication is concurrent with computation, so the length of the critical path does not change.

Summary

The dataflow execution pattern exploits instruction-level parallelism to improve performance and a reduced variance in each core's workload to improve energy efficiency.

The case studies above show that it is possible to use dataflow execution to improve both performance and energy consumption, but not by as much as might be expected. Performance improvements are due to the exploitation of instruction-level parallelism, and are limited by the length of the critical path; spreading the critical path across multiple cores adds network latency and increases execution time. Energy improvements are mainly due to having cores executing very small sections of code which fit entirely in the efficient instruction buffer; executing a single instruction on each core in order to bypass much of the pipeline did not always show clear benefits.

Upon further inspection, it was found that writing to and reading from a local register file consumed 2.0pJ, whereas sending a value to another core cost 5.7pJ (including buffer accesses, and assuming 50% of bits toggle). This difference of 3.7pJ is greater than the best case of 2.7pJ saved in the pipeline when a core is in dataflow mode, meaning that it is possible for dataflow execution to reduce energy efficiency. The cost of sending to another core is dominated by network buffers (39%) and the core-to-core network (43%). Section 6.4.5 explores how these overheads can be reduced to allow the dataflow execution pattern to further reduce energy.

The situation can also be improved by allowing a core to do more work but still bypass much of the pipeline. This can be achieved either by allowing the core to bypass functionality even when it is assigned multiple instructions (Section 6.4.3), or by adding or modifying instructions so that they can be used alone in the places where multiple instructions were needed previously (Section 6.4.4).

Spreading the critical code path across multiple cores can add network latency and reduce performance. Eliminating this additional latency by introducing more direct communication links is explored in Section 6.4.6.

From these case studies, it appears reasonably common to have one core persistently updating a value held in a register, and return the value when the loop completes. Even though only one register is accessed, the whole register file must be kept active. This could be improved by using explicit operand forwarding or a small operand register file [16]. Some architectures also provide functionality to automatically increment the value in a register, without the need for an additional instruction [11, 90].

6.4.3 Dataflow within a core

The costs of communicating between cores is one of the main factors which determines how effective dataflow execution can be when using tightly-coupled cores. Putting multiple instructions on a single core reduces the communication required, but means that the core cannot enter the low-power dataflow mode because the pipeline needs to alternate between the available instructions. This section explores allowing a core to enter dataflow mode even when it is assigned multiple instructions.

Since most of the energy benefits of dataflow execution are gained from bypassing much of the fetch and decode stages, the aim is to allow multiple instructions per core without reac-

tivating these components. It would be possible to have a small cache of decoded instructions and cycle through them, but this is likely to be relatively expensive, negating the benefits of bypassing other parts of the pipeline. An alternative is to add extra, optional, execute pipeline stages containing additional ALUs, and allow each ALU to be assigned a single operation for a long period. This approach is explored here.

The cost of implementing this scheme is dominated by the additional ALUs and pipeline registers. Each ALU has an area of $2862\mu m^2$ and each pipeline register has an area of $295\mu m^2$ (6.4% and 0.7% of the base core area, respectively). Extra (de)multiplexers are also required to steer data to and from the optional pipeline stages; it is expected that the cost of these is negligible.

Benefits of this approach include the ability for more cores to enter dataflow mode (in some cases it is impossible to reduce the workload to a single instruction; examples are discussed below), and the reduced costs of communication between ALUs in a single pipeline: data does not need to touch the register file or the network – only a much-cheaper pipeline register.

In order to avoid increasing the number of ports on the register file or input buffer array (which would have a detrimental cost to the common case when the extra ports aren't needed), the number of input operands which can be retrieved from each of these sources is limited to two, and only one output is collected. A single immediate operand can also be received from the decoder, as well as any data on the bypass network. This is enough data to support several operations simultaneously, but to simplify this study, I explore only one additional ALU and leave the exploration of more complex arrangements to future work.

Two ALUs are enough to allow a core to enter dataflow mode even when it has a pair of inseparable instructions, and also stops two-cycle operations such as multiplications and memory stores from being bottlenecks. (Note that some inseparable instruction pairs, such as a comparison followed by a branch, would not be helped by this technique because the branch completes in the decode pipeline stage and does not make use of an ALU. The instruction set modifications explored in Section 6.4.4 would be required for this case.)

The *crc* and *bitcount* benchmarks were mapped to an architecture capable of entering dataflow mode with two instructions on each core. The cores responsible for loading data from memory could not enter dataflow mode as they produced two outputs each: one message sent to the L1 cache to request data, and one message to send the retrieved data to the appropriate core.

Placing twice as many instructions on each core means that in the best case, half as many values need to be sent across the network, and instead can use a much-cheaper pipeline register.

Energy consumption for *crc* reduces by 15% over the baseline dataflow case: 43% of this is due to reduced network access, and 57% is due to reduced pipeline activity where more cores are in dataflow mode. Performance is unchanged. *bitcount* sees no improvement as there is no less communication than when each core can only execute one instruction in dataflow mode.

Having multiple ALUs in a single core can provide wider-spread benefits than those described in this section. Aside from the common approach of allowing multiple instructions to be issued each clock cycle, She et al. [118] add an extra ALU to a RISC pipeline to allow acceleration of common instruction pairs. A lookup table is added to the decoder, and a small number of opcodes are reserved to access entries of the table. The table contains all necessary control signals for both ALUs, and can be reprogrammed at runtime to best match the instruction pairs present in the application. For their processor, dynamic instruction count could be reduced by 25% and energy fell by 16% on average due to fewer register file and instruction cache accesses.

```
ldw 0(r3) -> 1
or.eop r0, r2, r0 -> 2
```

Figure 6.19: A bottleneck packet in the *crc* benchmark.

Park et al. [103] describe a technique for dynamically fusing a series of up to four low-latency operations on a CGRA so that they all complete within a single clock cycle. It may be possible to apply a limited form of this optimisation to Loki, so that if the two operations can complete in series, the intermediate pipeline register is bypassed and both operations complete in one cycle. This would save the energy consumed by the pipeline register, and decrease the latency of the operations. If combined with She's technique, this could be used to reduce the latency of longer sequences of sequential code, which is beneficial in all cases.

6.4.4 Reducing bottlenecks

The rate at which data can move through a dataflow network is limited by the slowest core on the critical path, and in the case of loops, the total length of the critical path. In some cases, spreading code across multiple cores can require extra instructions or network communications to be added, extending the length of the critical path and reducing performance. It can also be impossible to reduce a core's workload to a single instruction due to dependencies between instructions, so the core becomes a bottleneck. This section explores ways in which these bottlenecks can be reduced or eliminated.

As well as reducing the total number of instructions executed, which saves energy and time, these techniques allow more cores to enter a low-power dataflow mode, further reducing energy consumed.

In the case studies above, two main bottlenecks can be identified: memory access and control flow. These are addressed separately below.

Memory access

Loki's decoupled memory access means that loading data from memory is a two-stage process:

1. Send request to memory
2. Receive data from input buffer

The receipt of data can often be performed with zero overhead, since it is treated as an ordinary register read. However, when attempting to get down to one instruction per core, there are no other instructions which use the data, so reading from the buffer requires a separate instruction (Figure 6.19). Further, this puts a round trip to memory on the critical path of this dataflow node, which lengthens its runtime by at least two cycles.

In some cases, it may be possible to completely eliminate this memory access by instead making use of the core's local scratchpad memory (see Section 3.2.6) – this is the case in Figure 6.19, as the table being accessed in *crc* consists of 256 words.

In other cases, this may not be possible, and a different solution is required. Fortunately, Loki's network system provides this solution. Recall from Section 3.4 that each L1 cache bank has multiple virtual channels which are configured to send any requested data to a particular network address when cores first connect. This feature can be used to tell the memory bank to send data directly to the target core, rather than to the core which issued the request.

Care needs to be taken with this approach: if the target core stalls, it is possible for its input buffers to fill, which in turn will result in the memory bank stalling until there is space to send its data. This then prevents other memory operations progressing at that memory bank, which may result in deadlock. This problem can be avoided by ensuring that no other cores access the memory bank at the same time, and can be implemented either through careful placement of data in memory, or by using the memory configuration abilities mentioned in Section 3.4 to create a separate virtual cache which only one core has access to. In this section's evaluation, it is assumed that there are no deadlock issues.

The baseline for *crc* is the 8 core mapping (Figure 6.15c). Cores enter dataflow mode if they have been assigned only a single instruction.

Making use of these optimisations to memory access greatly improves performance. Having memory banks send data directly to another core allows both instances of `or.eop` to be removed, along with the core-to-core communication latency. This reduces the critical path latency from 13 cycles to 11, reducing inner loop execution time by 15% and energy by 13%.

Storing crc_32_tab in one of the core's scratchpads provides further benefits: as well as eliminating an expensive memory access, it allows three instructions to be removed which were previously responsible for computing the memory address (all of which were on the critical path). This brings the critical path latency down to six cycles: three for computation and three for communication. Combined with the memory access optimisation, execution time reduces by 54% and energy consumption improves by 33% over the baseline.

Placing the entire critical path on a single core reduces the critical path to three cycles, bringing execution time down to 23% of the baseline (a speedup of 2× over the 5-core implementation optimised for performance in Figure 6.15b). Energy consumption doesn't change much (it decreases slightly), showing that the benefits of putting more cores in low-energy dataflow mode do not outweigh the costs of increased communication. Section 6.4.6 explores introducing very cheap communication links to make it possible to have short critical paths and lots of cores in a low-energy state simultaneously.

While the optimisations in this section have significant benefits in their own right by reducing the amount of work required to execute a program, these benefits are amplified when used in used in combination with the dataflow execution pattern. Since dataflow uses instruction-level and pipeline-level parallelism to reduce the execution time of a loop iteration to the length of its critical path, any reduction in this has a proportionally larger effect on the overall execution time than when applied to sequential code.

Control flow

Infinite loops (via the *fetch persistent* instruction) are used widely in the dataflow execution pattern to remove the need for a branch instruction. At least one core's loop must be finite, however, so it can determine when the loop ends. This results in a core which has (at least) an instruction to determine whether the loop has ended, and a branch instruction. This is illustrated in Figure 6.20a.

This situation can be addressed by providing new versions of the *fetch persistent* instruction which check the value of the predicate register to determine whether the loop body should be repeated. These are shown in Table 6.1.

Since the predicate is computed during the execute pipeline stage, but instructions are retrieved in the fetch pipeline stage, this would introduce two pipeline bubbles if implemented naively. Instead, it is speculated that the loop body will always be repeated, and the pipeline is flushed when the predicate register holds the terminating value. Since it is impossible for any of

```
setnei.p r0, r3, 0 -> 3
ifp?ibjmp -4
```

(a) Bottleneck packet. The *ibjmp* instruction makes use of the predicate register, so cannot be separated from the instruction which writes to it.

```
setnei.p.eop r0, r3, 0 -> 3
```

(b) Resolved bottleneck. By making use of the new *fetchpstp* instruction, the conditional jump instruction is not needed.

Figure 6.20: A bottleneck packet in the *bitcount* benchmark.

Mnemonic	Description
fetchpstp	Fetch persistent while predicate is true
fetchpst!p	Fetch persistent while predicate is false

Table 6.1: New fetch instructions.

the flushed instructions to have reached a point in the pipeline where they have changed register state or sent data on to the network, the simple flush is sufficient. It is possible for an instruction to have performed a destructive read from an input channel, but any data received after the loop had finished must have been computed speculatively and can be safely discarded.

Figure 6.20b shows how the persistent instruction packet can be simplified with the help of the new instructions. The packet now has only one instruction and so is no longer a bottleneck, and can enter the low-power dataflow mode.

This optimisation is of little help in the *crc* benchmark, as the control core is not on the critical path. Removing instructions from the code path will never hurt, however, and energy consumption drops by a few percent. The technique is more useful in the *bitcount* benchmark, where the control core is on the critical path. Removing the branch instruction reduces the critical path from 4 cycles to 3, and this is reflected in the execution time, which also drops by 25%.

Summary

The techniques described in this section can be used to improve performance by reducing the duration of a sequence of instructions. This can be done either by eliminating superfluous communications over the network, or by reducing the number of instructions executed.

The optimisations are not exclusive to the dataflow execution pattern: they will be useful in any sequence of code which can make use of them. Their advantages are amplified when making use of dataflow, however, because dataflow graphs expose the critical paths of code sequences, which become bottlenecks to further improvements in performance. Since the critical path is often shorter than the whole code sequence, any reduction in its length will usually have a proportionally larger impact on overall execution time.

As well as improving performance of applications, reducing the number of instructions executed will result in lower energy consumption and improved caching behaviour. When making use of the dataflow execution pattern, there are additional energy benefits from being able to reduce cores' workloads to a single instruction.

The instruction set is one of the main features which determines how effective dataflow execution can be: a denser encoding allows cores to do more work while remaining in the efficient dataflow mode. A move to a full CISC instruction set may be excessive due to the

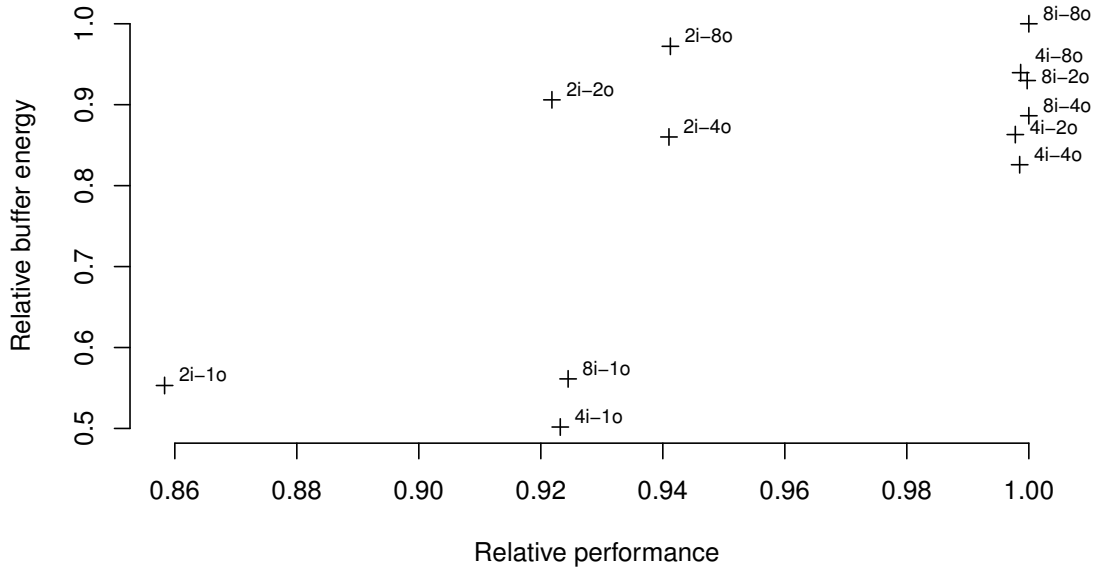


Figure 6.21: Network buffer design space exploration. Each point is labelled with the number of entries in the **i**nput and **o**utput buffers of each core and memory bank. Figures are relative to the 8i-8o implementation.

extra logic required to decode and execute the complex instructions, but selectively adding more-complex instructions may be beneficial. Examples include ARM's barrel shifter, which allows the second operand of almost any instruction to be shifted or rotated by any amount in addition to performing the main operation [11], and auto-incrementing registers [11, 90].

The only remaining bottlenecks are multiplication and store instructions as these take two cycles each.

6.4.5 Reducing power

It was found that the savings from greatly reduced switching activity in pipelines when cores entered dataflow mode did not generally make up for the costs of additional network communication. Network communication costs are dominated by network buffers (39%) and long wires (43%). This section explores ways in which these costs can be reduced to make dataflow a more profitable execution pattern.

Buffer optimisations

Figure 6.21 shows how performance and network buffer energy consumption vary with buffers of different sizes. Input and output buffer sizes were varied independently. Figures are relative to the largest buffers of 8 entries, and are averaged across all dataflow and task-level pipeline benchmarks. There was no clear difference in behaviour between the two execution patterns.

There is a large drop in energy consumption of almost 50% when output buffers have only a single entry – there is no need for any selection logic or multiplexers. Unfortunately, this drop in energy consumption also comes with a drop in performance as the single buffer space provides insufficient decoupling between communicating cores. Buffers with multiple entries are all much closer to each other in both energy and performance. Interestingly, there is a performance penalty when using input buffers with two entries, but not with output buffers of

two entries. This is because when using a task-level pipeline where multiple items need to be sent between stages, small input buffers result in data having to wait in the output buffer of the previous stage, which blocks any other network communications that might be required.

Overall, the best buffer implementations appear to be 4-entry input buffers and 4-entry output buffers, as they consume almost 20% less energy than their 8-entry counterparts with a negligible performance impact of less than 0.2%. I would expect an implementation with 2-entry output buffers to reduce energy even further without a significant impact on performance, but the current energy models suggest that a buffer with four entries consumes less energy than a buffer with two (Table 4.8).

The result of this exploration is that the default buffer parameters used previously are optimal: all buffers should have space for four words. This leaves network buffers consuming a large portion of total network energy, making fine-grained dataflow execution less viable. Optimisations which may help include: biasing the design towards the common case of writing to an empty buffer and reading from a buffer with a single item in it; allowing narrower values to be stored and retrieved without activating unneeded bit lines; or bypassing buffers entirely whenever possible.

Interconnect

The core-to-core network is one of the main additional contributors to energy consumption when using the dataflow execution pattern. This is because data needs to be communicated frequently, and since the network is capable of sending data to all cores simultaneously, the signal must be driven along the entire bus, wasting energy. Reducing the overheads of communication will make dataflow execution profitable in more situations, but will also help in non-dataflow cases. Three modifications to the network are explored:

1. Use of *intelligent repeaters* in the existing network [45]. These ensure that data is sent no further than necessary, so reduce the length of wire which switches. They also add to the latency of the link and consume a small amount of energy themselves, however, so there is a tradeoff between saving energy and keeping latency low.

2. Introduce a network for point-to-point communication between cores. The network used has the same design as the crossbar used between cores and memories. The crossbar is more efficient, but is an extra option for where data can be sent or received, so will result in larger, higher-energy multiplexers at the network interfaces which will be detrimental to all network communications.

3. Introduce direct communication links between neighbouring cores. The restriction of a single destination makes this the most efficient option, but it may not be possible to use all of the time. Again, larger multiplexers at the network interfaces are required to allow this option.

A summary of the energy consumption of these options is given in Table 6.2. For this study, many of the potential restrictions are removed to show the limits of how effective each type of network can be: it is assumed that there are enough intelligent repeaters to ensure that signals travel the minimum possible distance; larger multiplexers are no more expensive; and there are direct links between all pairs of cores. Some of these restrictions are addressed after the initial evaluation.

Approach	Energy in pJ/bit
Default	0.151
Intelligent repeaters	$distance\times$ 0.151
Crossbar	$distance\times$ 0.048 + 0.084
Direct links	$distance\times$ 0.057

Table 6.2: Comparison of energy consumption of different communication networks. All distances are in millimetres. The overheads of intelligent repeater logic and larger multiplexers are ignored for this limit study.

In order to increase communication distances and avoid trivial communication patterns, the dataflow mappings involving the most cores were selected from the case studies: an 8-core *crc* mapping and a 4-core *bitcount* mapping.

Energy of networks with intelligent repeaters was estimated by computing the average length of wire which would be switched for each repeater layout, and scaling the energy of the core-to-core network accordingly.

Nearest neighbour communication is not as much of a restriction as it may first seem. The worst case is where all instructions receive two inputs and send an output over the network. Since each core has only two neighbours, it is possible to put up to two of these three communications on direct links. In practice, however, it is usually possible to send more than $\frac{2}{3}$ of point-to-point messages between neighbours. Indeed, 75% of *bitcount*'s communication is between neighbouring cores, and 88% for *crc*. There are several reasons for this:

- Some cores need to multicast their results, so use the existing core-to-core network and do not need to be placed next to the recipients of the data.
- Some cores communicate with memory instead of other cores.
- Some instructions use operands which do not need to be received over the network – they are constants or forwarded internally.
- Some instructions have fewer than two inputs and one output.

These local links also have the advantage that they do not need to be limited by tile boundaries: it would make sense to connect the last core of one tile to the first core of the adjacent tile. This would increase intertile bandwidth, and allow dataflow networks to span multiple tiles more easily. It would be possible to provide direct links in four directions, forming a 2D mesh network, but the current floorplan means that the distances between cores vary widely depending on the direction travelled. Only the shortest links are explored here.

Very cheap nearest-neighbour communication is also useful in other execution patterns: DOACROSS where loop iteration i is dependent on iteration $i-1$; pipeline parallelism; and compiler-managed instruction-level parallelism.

Figure 6.22 shows how the energy consumption of network communication changes with the different optimisations. Three implementations of a multicast network with repeaters are explored: one repeater splits the bus in half, so the minimum length of wire switched is 0.5mm; more repeaters further reduce the length of wire switched. Using repeaters has a large impact on the energy consumption of the core-to-core interconnect, suggesting that most communication is between cores which are physically close to each other. Using a dedicated point-to-point

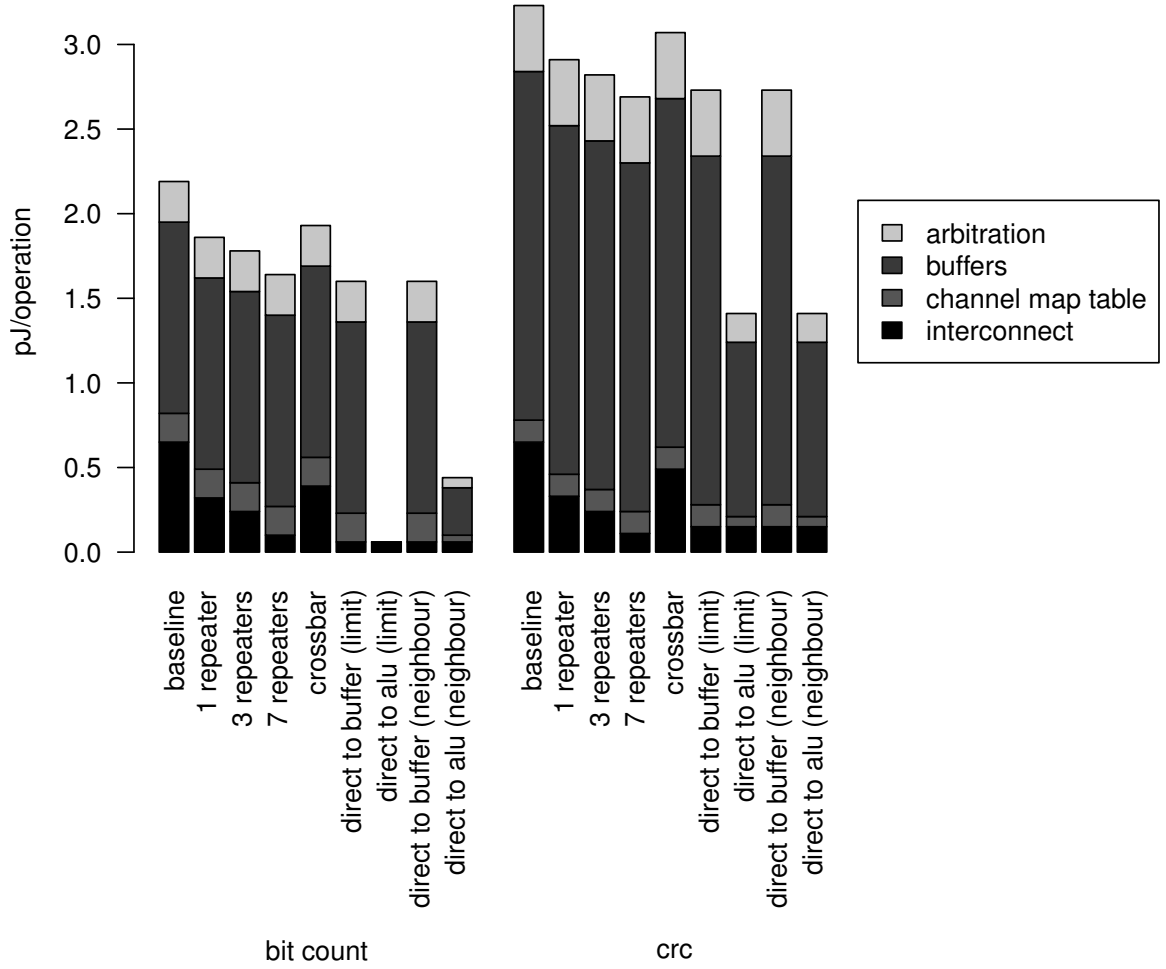


Figure 6.22: Energy comparison for different core-to-core interconnect options. *limit*: direct links between arbitrary cores; *neighbour*: direct links between adjacent cores only.

crossbar reduces energy consumption, but not by as much as a single repeater on a multicast bus. Multiple implementations of direct connections between cores are explored: those which deliver data directly to a particular input buffer of the target core and those which deliver data directly to the input of the ALU using an extension to each pipeline's bypass network. The *limit* implementations allow direct communication between arbitrary cores, while the *neighbour* implementations restrict direct communication to cores which are next to each other. Direct communications almost completely eliminate the energy consumed by interconnect, and when sending data directly to an ALU, other parts of the network, such as arbiters and buffers, are used much less.

Direct links between ALUs have been explored previously. SIMD-Morph [34] is a SIMD architecture which allows groups of 4 functional units to forward data between each other arbitrarily in order to accelerate dataflow subgraphs in code regions lacking data-level parallelism. This differs from a superscalar forwarding network in that any instruction can be executed in SIMD mode, and multiple operations can be executed in series in a single clock cycle. It is also possible to forward data from one group to a neighbouring group. This rich communication structure allows sequential code to be accelerated by an average of 2.6×. Having such a large number of potential operand sources requires a crossbar within the pipeline, which we expect to be too large, slow and energy-hungry for Loki's purposes, however. It is also shown that only about 1% of dataflow subgraphs in a range of benchmarks (including MiBench) have more than 8 operations, so Loki's tile size should be sufficient.

Summary

While the baseline dataflow implementation on Loki was able to boost performance of application kernels, energy consumption often increased when more cores were used, despite the reduction in pipeline activity. This was because of a great increase in network activity. Most of the energy consumed in the network is in network buffers and long wires, so this section explored optimisations to these. It was found that buffer sizes were already optimal, but core-to-core interconnect energy could be reduced by 80% (*crc*) and 95% (*bitcount*) by providing direct links between neighbouring cores.

As a result, the average total cost of sending data to a neighbouring core reduces from 5.7pJ to 3.4pJ. This is only 1.4pJ more expensive than storing and retrieving a word from a local register file, so can be worthwhile if an additional core is able to enter dataflow mode (up to 2.7pJ savings per instruction).

It should be noted that it is also possible to optimise cases where data is kept on the same core, which could once again increase the overheads of dataflow execution. Such optimisations include providing another level of data storage hierarchy which is even smaller and cheaper than a typical register file [16], and providing dedicated register ports for small values [7]. The optimisations described in this section would still be useful on an architecture which uses the network as much as Loki.

6.4.6 Reducing latency

One of the main limiting factors of dataflow execution on the Loki architecture is that communication between cores can add latency to the critical path, worsening performance. This means that the best mapping of the application often involves putting the entire critical path on a single core. While it is possible to improve performance and reduce energy consumption in this way, the full potential of dataflow parallelism is not being realised. This section explores the effects

of zero-cycle communication between neighbouring cores. This makes use of the direct links described in the previous section.

The latency of a channel has many determining factors, including its length, wire spacing, voltage and metal layers used. For this experiment, I keep a constant length of 125µm (the width of a Loki core), a voltage of 1.1V, and the metal layers used by the synthesis tools.

When the synthesis tools are configured to minimise latency, it takes 39ps for a signal to travel from one end of the channel to the other, at a cost of 10fJ per bit toggled. When configured to minimise energy consumption, the latency is 85ps, at a cost of 7fJ per bit. When compared with Loki's clock period of 2300ps, it seems likely that either option could be implemented without significantly extending the critical path – I select the lower-energy option for these experiments.

Using this zero-cycle communication latency, it is possible to spread the critical code path of an application across multiple cores without extending its execution time. Performance for both benchmarks was found to match the implementation with the entire critical path on one core (see Figures 6.16 and 6.18), but energy consumption was lower since more cores were in dataflow mode (Figure 6.22).

This optimisation allows Loki to achieve the perfect 5-tuple in Taylor et al.'s network taxonomy [125] (discussed in Section 2.3.1). Loki's various components of network latency are now <0,0,0,0,0>, the same as a superscalar architecture, suggesting that instruction-level parallelism could be exploited efficiently in sequential code. Superscalar architectures do still have an advantage, however: their zero-cycle latency networks typically have a higher connectivity, and functional units are able to forward data to more than just their nearest neighbours.

A further advantage of direct links between neighbouring cores, as mentioned previously, is that they can provide bridges between neighbouring tiles without needing to go through the relatively expensive global network.

6.4.7 Parallelism extraction

Generating a dataflow graph automatically is simple. Indeed, it is commonly used in compilers as an intermediate data structure. The main challenges in automatically exploiting dataflow parallelism on Loki then become:

1. Determining how the dataflow graph should be mapped across multiple tiles. Inter-tile bandwidth and latency are not as good as on the intra-tile network, so partitioning algorithms will be needed to cut the graph at the best points. This is a common problem for tiled architectures, and tools exist to generate sensible mappings automatically [96].

2. Determining how many instructions to execute on each core. Going to the extreme of one instruction per core can result in large energy reductions, but can also increase the length of the critical path and degrade performance unless some of the suggested optimisations are applied. A tradeoff will need to be made, depending on the requirements of the program.

Alternatively, the instructions could be automatically distributed across cores using a technique similar to instruction-level distributed processing [30] or TRIPS/EDGE [28], where a program is separated into *strands* of dependent instructions with minimal communication between strands.

Architecture	FUs	Max. issue rate	Local memory	Area/mm^2	
				Memory	Total
ARM A15 [41, 74]	8	3	32+32kB	0.5*	4.5*
ADRES [24]	16	16	0.5kB	0.11	0.31
Loki	8	8	64kB	0.5	1.0
TRIPS [114]	16	16	80+32kB	3.8	8.9

* Memory area estimated using models from Section 4.5.

Table 6.3: Comparison of architectures capable of dataflow execution. All areas are for architectures scaled to a 40nm process and represent one core or tile of the architecture. L2 caches are excluded from area figures.

6.4.8 Conclusion

Dataflow is another option available to the programmer and compiler, and has the potential to improve both performance and energy consumption. It works in a wide variety of situations, but is best-suited to tight loops. The dataflow execution pattern allows an application to be optimised when no other forms of parallelism are available. Dataflow execution can require huge numbers of cores to achieve its full potential, but on an architecture such as Loki, where cores are abundant, its use may still be worthwhile.

Like some CGRAs, Loki offers the ability to map multiple instructions to each core to reduce the resources required [100]. The energy efficiency of the core degrades gracefully as the number of instructions on it increases and more of the pipeline is required. Loki lacks the speculation hardware used by architectures such as Multiscalar to execute multiple blocks of instructions simultaneously [39], as the overheads of logic required to make predictions and roll back when a prediction fails were considered too high. Instead, Loki offers a more-flexible communication network, allowing the blocks to be smaller and more-tightly coupled without impacting performance. Some form of speculation may still be required to make the most of the Loki architecture when ILP is limited; this is a subject of future work.

Table 6.3 compares a selection of architectures which are capable of dataflow execution. The ARM core is able to use its large issue window to dynamically schedule instructions to its limited resources, ADRES is a coarse-grained reconfigurable array optimised to accelerate tight loops, and TRIPS makes use of an instruction set with a dataflow representation. The ARM core's dynamic execution means that it is not practical to issue an instruction to every functional unit each clock cycle: the overheads of finding enough independent instructions of the right types would be too high. It is able to achieve relatively high performance on sequential code, however. ADRES, being an accelerator to a more-traditional processor, is able to have a very specialised design with little memory and densely packed functional units. TRIPS's relatively large area reflects its focus on floating point operations and multiple copies of complex structures such as load-store queues. Loki, with its 8 functional units per square millimetre, represents a compromise between the general-purpose ARM and TRIPS (2 FUs per mm^2) and the specialised ADRES (50 FUs per mm^2), while still being capable of general purpose computation and other forms of parallelism. Interestingly, the ARM A15 has around 40 entries in its issue window; in a similar area, it would be possible to place five Loki tiles and dedicate an entire core to each instruction in the window (and have five times the L1 cache capacity), giving a potential for much greater parallelism.

Optimisation	Execution time impact	Energy impact
Two functional units per core	0%	-8%
Memory configuration	-8%	-7%
New instructions	-12%	-19%
Direct links between neighbouring ALUs	-16%	-24%

Table 6.4: Summary of optimisations to the dataflow execution pattern. Figures are averaged over the two case studies, even when the optimisation only applies to one. In addition to the energy and performance impacts, adding an extra ALU increases each core's area by 7%.

Very fine-grained dataflow (one instruction per core) does not show a clear benefit on the base Loki architecture: the energy and delay associated with communication, while very small compared with other general-purpose architectures, still offset any benefits of improved caching and reduced switching. Several optimisations were explored to reduce these overheads:

- Placing multiple functional units in each core increased the number of cores which could enter the low-power dataflow mode and reduced communication costs between operations on the same core.
- Memory banks were configured to send data directly to the target core.
- New variations of the *fetch persistent* instruction were introduced to allow tighter loops.
- Interconnect was added or modified to reduce the length of wire which switches when cores communicate. In the case of direct links between neighbouring cores, it is also expected to be possible to reduce communication latency to zero cycles.

Table 6.4 summarises the effects of these optimisations. With the assumption that larger multiplexers have a negligible overhead, none of the optimisations result in degradation of performance or energy efficiency for either benchmark. Furthermore, the optimisations can be applied in combination for additional improvements – they are all independent, but the effects of some overlap. For example, adding an extra functional unit to each core will result in more information being kept within the core and reduce the amount of network communication, reducing the impact of cheaper core-to-core communication.

Although these modifications were suggested with the dataflow use case in mind, they all have the potential to help in other situations too. It is hoped that by making the optimisations applicable in a more general situation, they avoid becoming an overhead to the common case.

6.5 Summary

This chapter has shown that efficient communication between processor cores and a flexible, general-purpose architecture make it possible to exploit many different forms of parallelism. It is possible to tailor the architecture to the program by taking advantage of the types and amounts of parallelism available. The structure of execution patterns can be hierarchical – it is possible to set up many parallel pipelines, or exploit dataflow within a pipeline stage, for example.

Since any core can work independently, or as a part of any of the execution patterns explored here, a door is opened to the realm of dynamic reconfiguration. It would be feasible to adjust

the size of parallel structures at runtime, or switch from one execution pattern to another based on performance metrics. This is left for future work.

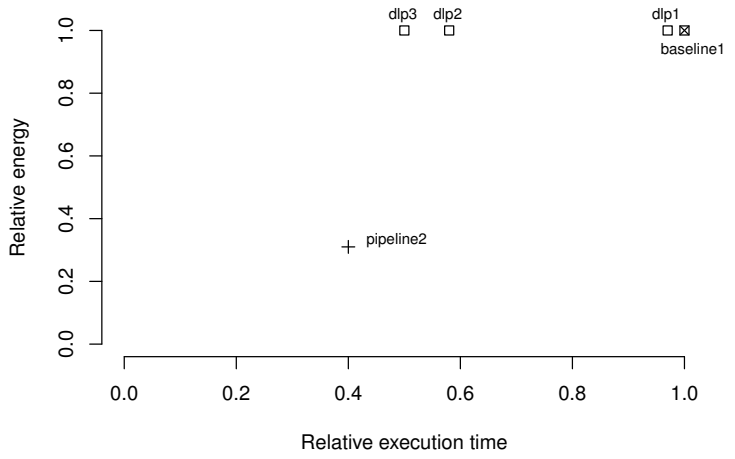
A spectrum of approaches is available to reduce instruction supply energy. Task-level pipelines reduce the amount of code executed on each core, potentially to a point where the small L0 caches can hold the entire active code region. Going further, if the code executed on each core can be reduced to 16 instructions or less, the cheaper instruction buffer can be used instead, and the pipeline begins to look more like a dataflow network. (Of course, an intermediate stage of some cores using their caches and some using their buffers is also possible.) In addition, the cache pinning technique (Section 5.1.3) can be used to bridge the gaps in this spectrum by putting as many instructions as possible in the cheap-to-access cache or buffer, and fetching all remaining instructions from another source. Finally, for tight loops where it is possible to reduce some cores' workloads to a single instruction, the entire fetch/decode section of the pipeline becomes inactive.

Orthogonal to these approaches is the data-level parallelism execution pattern, which is able to increase performance by executing multiple copies of the same code simultaneously. Energy can also be reduced by eliminating redundant work between cores. It is possible to dedicate a single core to providing common data to all other cores, and cores can share their instructions to make better use of their aggregate cache capacity. Simple modifications to the DLP pattern can allow programs with loop-carried dependencies and loops with highly variable iteration execution times to be accelerated, though the instruction sharing mechanism is not possible in these cases.

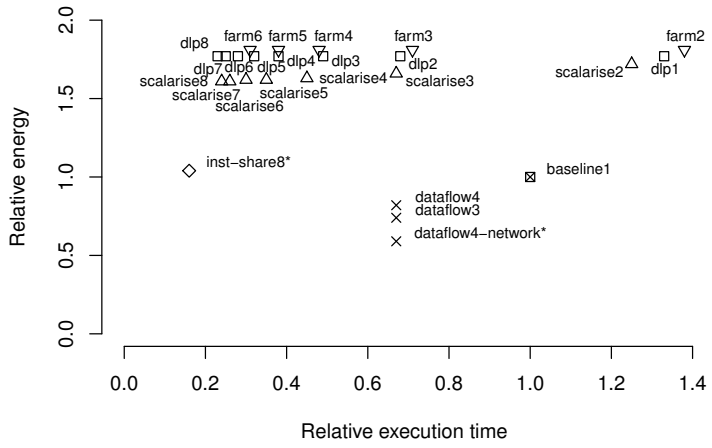
Also possible are compiler-managed techniques to exploit instruction-level parallelism across multiple cores. These include emulating a VLIW-like architecture, using the low-latency network for data forwarding, or assigning decoupled instruction strands to each core [30].

Figure 6.23 presents a comparison of execution patterns for each of the nine benchmark kernels used in this chapter. It can be seen that there is no execution pattern which is always the best: different patterns are useful in different situations. Indeed, there is often a selection of configurations for a particular benchmark which lie on the Pareto front, allowing for different energy-performance tradeoffs. This confirms that flexibility is an important feature of the Loki architecture, as it allows better application implementations than if only a subset of parallelism types were possible.

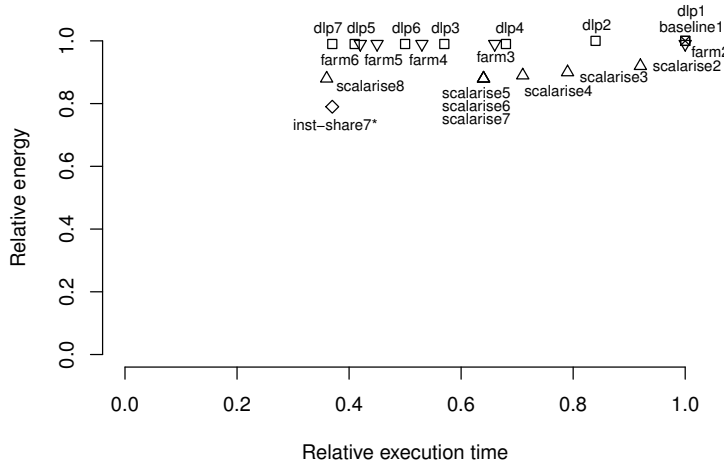
The flexibility of the hardware puts Loki in an unusual situation in terms of deciding the requirements of individual components such as caches and register files. Many potential deficiencies in the hardware, such as a lack of cache capacity, can be mitigated by simply making use of more cores and adding in core-to-core communication where necessary. To this end, it may be profitable to deliberately make the hardware structures too small to be useful by themselves (since smaller structures are cheaper to access), knowing that many will be combined at runtime to execute the program. At some point, the costs of additional communication are going to outweigh the benefits of smaller and cheaper structures, but the point at which this happens is dependent on the goals of the program. Work has been done on dynamically resizing such structures [8], and this could be useful as it would allow the programmer or compiler (or even a run-time system) to choose where on the spectrum a particular program should lie, instead of it being decided by the hardware designer.



(a) ADPCM encode

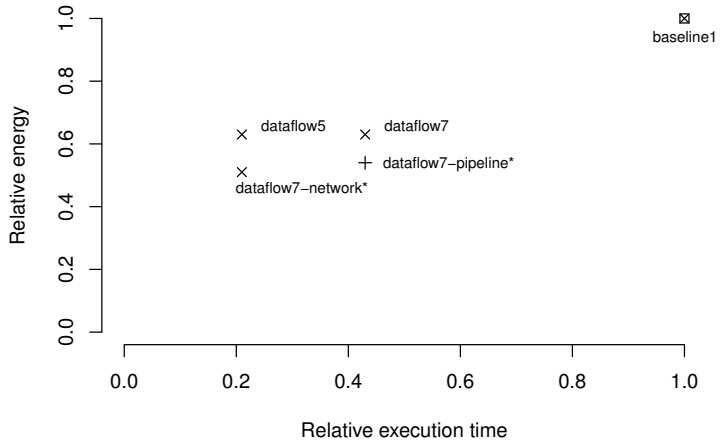


(b) Bit count inner loop

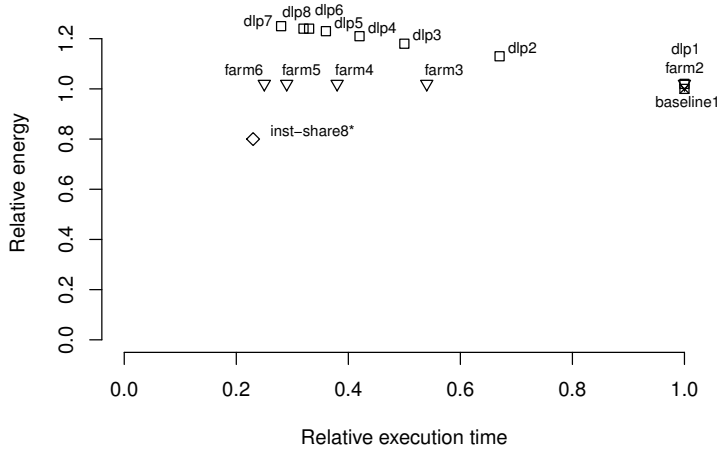


(c) Bit count outer loop

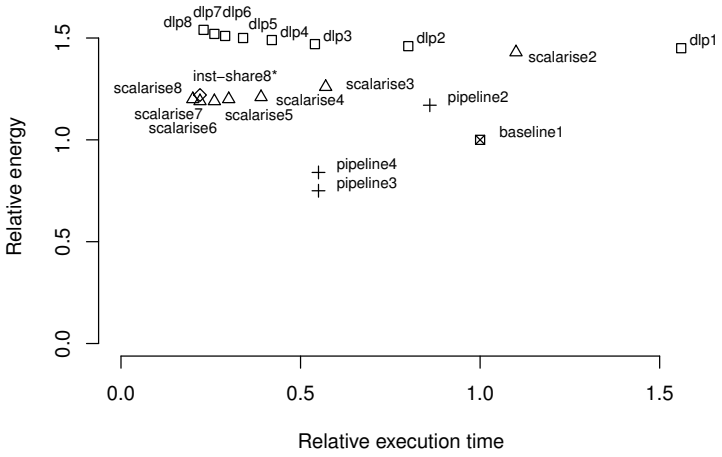
Figure 6.23: Summary of execution patterns. All results are relative to a single-core baseline. *dlp*: data-level parallelism pattern; *scalarise*: DLP with common work extracted to a helper core; *farm*: DLP using a worker farm for load balance; *inst-share*: DLP with instruction sharing; *pipeline*: task-level pipeline execution pattern; *(continued on next page)*



(d) CRC

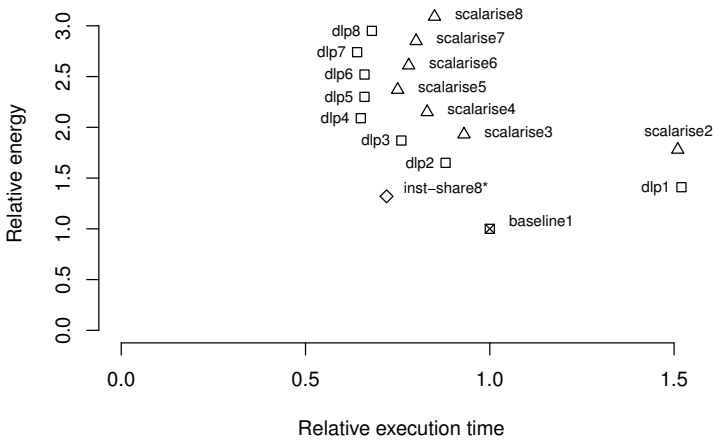


(e) Dijkstra's algorithm

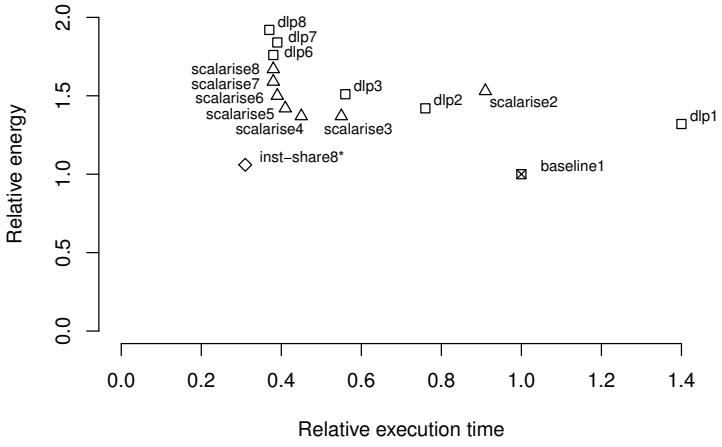


(f) JPEG colour conversion

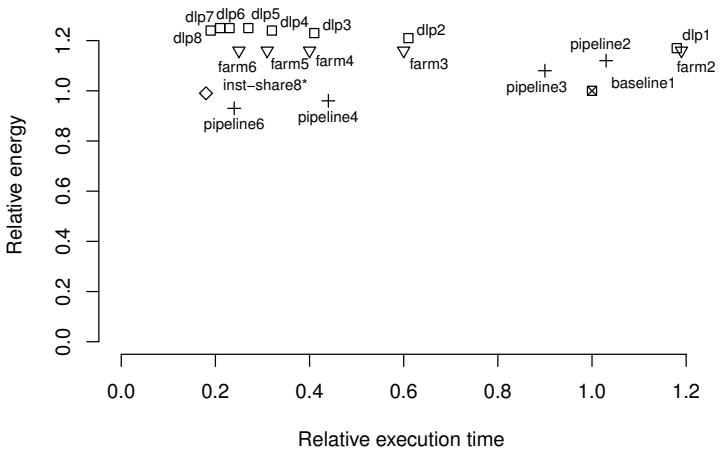
Figure 6.23: *(Continued) dataflow*: dataflow execution pattern; *dataflow-network*: dataflow with direct links between neighbouring cores; *dataflow-pipeline*: dataflow with two ALUs per core; <execution-pattern>N: pattern used a total of N cores; <execution-pattern>*: results are estimated from a limit study, rather than a simulation. *(Continued on next page)*



(g) JPEG DCT



(h) JPEG Huffman encoding initialisation



(i) String search

Figure 6.23: *(Continued from previous page)*

CHAPTER 7

CONCLUSION

In this dissertation, I have shown that providing an efficient and flexible communication mechanism between processor cores is sufficient to allow a wide range of parallel structures to be supported. Many of these *execution patterns* are capable of simultaneously improving performance and reducing energy consumption.

I introduced the Loki architecture, made up of an array of simple, low-power, homogeneous cores and a flexible network, which served as a testbed for exploring these different forms of parallelism. The streamlined processor pipeline allows energy per operation to be reduced to around 10pJ: an order of magnitude lower than typical mobile application processors.

For each of the main components of the Loki architecture, I developed a model of its energy, performance and area. These models were used to determine when and where energy was being consumed, allowing optimisation to be targeted at regions which stand to benefit the most. The models allowed me to perform a basic design space exploration to select a suitable baseline implementation of a single Loki tile. It is likely that this implementation will need to be tuned slightly as we come to better understand how cores are used together in practice.

Finally, I explored a range of execution patterns, each representing a different way of using multiple cores to execute a program. I explored data-level parallelism, task-level pipelines, and dataflow-style execution. Since these patterns are implemented in software, it is possible to tailor them to each application: for example, data-level parallelism can easily be modified to tolerate loop-carried dependencies or highly variable task lengths. I also suggested hardware modifications to improve efficiency when implementing particular execution patterns, and found that many of these optimisations would also help other patterns, or even sequential execution on a single core.

Cheap communication is an enabling technology. It introduces flexibility and allows for resource sharing, which can be used both to improve performance and reduce power consumption. Many different execution patterns can be used to exploit the parallelism available in a program, and these patterns can be combined arbitrarily and modified with little overhead. Efficient communication also opens the door to new techniques such as software transactional memory which were previously too expensive because of the communication overheads of traditional architectures. By increasing the coupling between cores even further so that pipelines can communicate directly with each other, it is possible to achieve further improvements.

Future mobile systems will be required to provide 1000GOPS at around 1pJ/operation in the presence of faulty transistors and spiralling engineering costs. This work is a step towards many-core systems with more than 1000 cores which will, I predict, be able to achieve this target without the need for complex heterogeneous architectures. I suggest that design and

verification effort is better spent on optimising a regular, all-purpose architecture, rather than a wide-range of programmable processors and fixed-function accelerators.

7.1 Future work

The design of the Loki architecture is still in its infancy, and there is still much work to be done. The following are just some of the possible avenues for further exploration.

- Generate a design for the global network, and explore how the execution patterns behave when mapped across multiple tiles.
- Operating system support: Loki currently lacks interrupts, virtual memory, and protection, all of which are near-essential for running a modern operating system. In some cases, hardware modifications will be required, but in others, it may be possible to reserve cores to perform the tasks in software.
- Optimise each of the individual components of the architecture. When implementing Loki, a deliberate decision was made to minimise the time spent optimising components, in favour of exploring the design at a higher level. Decades of work have been put into developing efficient microprocessor designs, so there is bound to be scope for improvement.
- Instruction set architecture exploration: Loki's ISA is based on that of MIPS, with only slight changes. A thorough analysis and comparison with other embedded architectures will allow any gaps to be identified and filled. Dally has stated that "a selective return to complex instruction sets" makes sense when optimising energy consumption [33], and from the results of this thesis, I agree.
- Revisit the design space exploration with the knowledge that cores will be used together to execute a program. It may be the case that resources are currently overprovisioned to allow satisfactory execution of a program on a single core, when this use case is unusual in practice.
- Develop automated techniques for recognising which types of parallelism are available in a program and generating the optimal virtual architecture. Automatic parallelisation is extremely difficult, but Loki's efficient core-to-core communication allows even tightly-coupled code regions to be split across multiple cores.
- Allow reconfiguration of more components, to increase the potential of software specialisation. A potential target for this is the network: configurable network topologies and protocols could allow optimised communication structures.

BIBLIOGRAPHY

[1] The OpenMP API specification for parallel programming. http://openmp.org.

[2] Transputer architecture. http://www.transputer.net/fbooks/tarch/tarch.pdf, July 1987.

[3] STi5100: Low-cost interactive set-top box decoder. http://www.bdtic.com/DownLoad/ST/STi5100.pdf, May 2004.

[4] June 2011 Green500 list press release. http://www.green500.org/printpdf/105, June 2011.

[5] S. Abramsky. Computational interpretations of linear logic. *Theoretical Computer Science*, 111(12):3 – 57, 1993.

[6] V. Agarwal, M. S. Hrishikesh, S. W. Keckler, and D. Burger. Clock rate versus IPC: the end of the road for conventional microarchitectures. *SIGARCH Computer Architecture News*, 28(2):248–259, 2000.

[7] A. Aggarwal and M. Franklin. Energy efficient asymmetrically ported register files. In *Proceedings of the 21st International Conference on Computer Design*, ICCD '03, pages 2–, Washington, DC, USA, 2003. IEEE Computer Society.

[8] D. H. Albonesi, R. Balasubramonian, S. G. Dropsho, S. Dwarkadas, E. G. Friedman, M. C. Huang, V. Kursun, G. Magklis, M. L. Scott, G. Semeraro, P. Bose, A. Buyuktosunoglu, P. W. Cook, and S. E. Schuster. Dynamically tuning processor resources with adaptive processing. *Computer*, 36(12):49–58, Dec. 2003.

[9] B. S. Ang and M. Schlansker. ACRES architecture and compilation. Technical report, Hewlett-Packard, April 2004.

[10] ARM Ltd. ARM1176 processor. http://www.arm.com/products/processors/classic/arm11/arm1176.php, 2013.

[11] ARM University Program. The ARM instruction set. http://simplemachines.it/doc/arm_inst.pdf, May 2013.

[12] K. Asanovic, R. Bodik, B. C. Catanzaro, J. J. Gebis, P. Husbands, K. Keutzer, D. A. Patterson, W. L. Plishker, J. Shalf, S. W. Williams, and K. A. Yelick. The landscape of parallel computing research: A view from Berkeley. Technical Report UCB/EECS-2006-183, EECS Department, University of California, Berkeley, Dec 2006.

[13] O. Azizi, A. Mahesri, B. C. Lee, S. J. Patel, and M. Horowitz. Energy-performance tradeoffs in processor architecture and circuit design: a marginal cost analysis. In *Proceedings of the 37th annual international symposium on Computer architecture*, ISCA '10, pages 26–36, New York, NY, USA, 2010. ACM.

[14] J. Balfour. *Efficient Embedded Computing*. PhD thesis, Stanford University, May 2010.

[15] J. Balfour, W. Dally, D. Black-Schaffer, V. Parikh, and J. Park. An energy-efficient processor architecture for embedded systems. *IEEE Computer Architecture Letters*, 7:29–32, 2008.

[16] J. Balfour, R. Halting, and W. Dally. Operand registers and explicit operand forwarding. *Computer Architecture Letters*, 8(2):60 –63, February 2009.

[17] C. Batten, A. Joshi, J. Orcutt, A. Khilo, B. Moss, C. Holzwarth, M. Popovic, H. Li, H. I. Smith, J. Hoyt, F. Kartner, R. Ram, V. Stojanovic, and K. Asanovic. Building manycore processor-to-DRAM networks with monolithic silicon photonics. In *High Performance Interconnects, 2008. HOTI '08. 16th IEEE Symposium on*, pages 21–30, 2008.

[18] D. U. Becker, D. B. Sheffield, and V. Parikh. Instruction compounding for embedded microprocessors. Class project, 2008.

[19] K. Bergman, S. Borkar, D. Campbell, W. Carlson, W. Dally, M. Denneau, P. Franzon, W. Harrod, J. Hiller, S. Karp, S. Keckler, D. Klein, R. Lucas, M. Richards, A. Scarpelli, S. Scott, A. Snavely, T. Sterling, R. S. Williams, and K. Yelick. Exascale computing study: Technology challenges in achieving exascale systems, 2008. Pages 130–131.

[20] N. Binkert, B. Beckmann, G. Black, S. K. Reinhardt, A. Saidi, A. Basu, J. Hestness, D. R. Hower, T. Krishna, S. Sardashti, R. Sen, K. Sewell, M. Shoaib, N. Vaish, M. D. Hill, and D. A. Wood. The gem5 simulator. *SIGARCH Comput. Archit. News*, 39(2):1–7, Aug. 2011.

[21] G. Blake, R. G. Dreslinski, T. Mudge, and K. Flautner. Evolution of thread-level parallelism in desktop applications. In *Proceedings of the 37th annual international symposium on Computer architecture*, ISCA '10, pages 302–313, New York, NY, USA, 2010. ACM.

[22] S. Borkar. Designing reliable systems from unreliable components: the challenges of transistor variability and degradation. *Micro, IEEE*, 25(6):10–16, 2005.

[23] S. Borkar. Thousand core chips: a technology perspective. In *Proceedings of the 44th annual Design Automation Conference*, DAC '07, pages 746–749, New York, NY, USA, 2007. ACM.

[24] F. Bouwens, M. Berekovic, A. Kanstein, and G. Gaydadjiev. Architectural exploration of the ADRES coarse-grained reconfigurable array. In *Proceedings of the 3rd international conference on Reconfigurable computing: architectures, tools and applications*, ARC'07, pages 1–13, Berlin, Heidelberg, 2007. Springer-Verlag.

[25] M. Boyer, D. Tarjan, and K. Skadron. Federation: Boosting per-thread performance of throughput-oriented manycore architectures. *ACM Trans. Archit. Code Optim.*, 7:19:1–19:38, December 2010.

[26] K. Brill. The economic meltdown of Moore's law and the green data center, 2007.

[27] D. Brooks, V. Tiwari, and M. Martonosi. Wattch: a framework for architectural-level power analysis and optimizations. In *Computer Architecture, 2000. Proceedings of the 27th International Symposium on*, pages 83–94, June 2000.

[28] D. Burger, S. W. Keckler, K. S. McKinley, M. Dahlin, L. K. John, C. Lin, C. R. Moore, J. Burrill, R. G. McDonald, W. Yoder, and the TRIPS Team. Scaling to the end of silicon with EDGE architectures. *Computer*, 37(7):44–55, July 2004.

[29] S. Campanoni, T. Jones, G. Holloway, V. J. Reddi, G.-Y. Wei, and D. Brooks. HELIX: automatic parallelization of irregular programs for chip multiprocessing. In *Proceedings of the Tenth International Symposium on Code Generation and Optimization*, CGO '12, pages 84–93, New York, NY, USA, 2012. ACM.

[30] I. Caulfield. *Complexity-effective superscalar embedded processors using instruction-level distributed processing*. PhD thesis, University of Cambridge, December 2007.

[31] J.-C. Chiu, Y.-L. Chou, and P.-K. Chen. Hyperscalar: A novel dynamically reconfigurable multi-core architecture. In *Parallel Processing (ICPP), 2010 39th International Conference on*, pages 277 –286, September 2010.

[32] E. S. Chung, P. A. Milder, J. C. Hoe, and K. Mai. Single-chip heterogeneous computing: Does the future include custom logic, FPGAs, and GPGPUs? In *Proceedings of the 2010 43rd Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO '43, pages 225–236, Washington, DC, USA, 2010. IEEE Computer Society.

[33] W. Dally, J. Balfour, D. Black-Schaffer, J. Chen, R. Harting, V. Parikh, J. Park, and D. Sheffield. Efficient embedded computing. *Computer*, 41(7):27 –32, July 2008.

[34] G. Dasika, M. Woh, S. Seo, N. Clark, T. Mudge, and S. Mahlke. Mighty-morphing power-SIMD. In *Proceedings of the 2010 international conference on Compilers, architectures and synthesis for embedded systems*, CASES '10, pages 67–76, New York, NY, USA, 2010. ACM.

[35] J. Dean and S. Ghemawat. MapReduce: simplified data processing on large clusters. *Commun. ACM*, 51(1):107–113, Jan. 2008.

[36] L. Djoudi, J.-T. Acquaviva, and D. Barthou. Compositional approach applied to loop specialization. *Concurr. Comput. : Pract. Exper.*, 21(1):71–84, Jan. 2009.

[37] H. Esmaeilzadeh, E. Blem, R. St. Amant, K. Sankaralingam, and D. Burger. Dark silicon and the end of multicore scaling. In *Proceedings of the 38th annual international symposium on Computer architecture*, ISCA '11, pages 365–376, New York, NY, USA, 2011. ACM.

[38] A. Fog. The microarchitecture of Intel, AMD and VIA CPUs. www.agner.org/optimize/microarchitecture.pdf, February 2012.

[39] M. Franklin. The multiscalar architecture. Technical report, University of Wisconsin-Madison, 1993.

[40] W. W. L. Fung, I. Sham, G. Yuan, and T. M. Aamodt. Dynamic warp formation and scheduling for efficient GPU control flow. In *Proceedings of the 40th Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO 40, pages 407–420, Washington, DC, USA, 2007. IEEE Computer Society.

[41] J. S. Gardner. MIPS Aptiv cores hit the mark. *Microprocessor Report*, May 2012, May 2012.

[42] M. Gebhart, S. W. Keckler, and W. J. Dally. A compile-time managed multi-level register file hierarchy. In *MICRO*, pages 465–476, December 2011.

[43] D. Greenfield. *Rentian Locality in Chip Multiprocessors*. PhD thesis, University of Cambridge, April 2010.

[44] P. Greenhalgh. Big.LITTLE processing with ARM CortexTM-A15 & Cortex-A7. Technical report, ARM, September 2011.

[45] B. Grot, J. Hestness, S. Keckler, and O. Mutlu. Express cube topologies for on-chip interconnects. In *High Performance Computer Architecture, 2009. HPCA 2009. IEEE 15th International Symposium on*, pages 163 –174, February 2009.

[46] M. Gschwind, H. Hofstee, B. Flachs, M. Hopkin, Y. Watanabe, and T. Yamazaki. Synergistic processing in Cell's multicore architecture. *Micro, IEEE*, 26(2):10–24, 2006.

[47] S. Gupta, A. Ansari, S. Feng, and S. Mahlke. StageWeb: Interweaving pipeline stages into a wearout and variation tolerant CMP fabric. In *Dependable Systems and Networks (DSN), 2010 IEEE/IFIP International Conference on*, pages 101–110, June 2010.

[48] S. Gupta, S. Feng, A. Ansari, B. Jason, and S. Mahlke. The StageNet fabric for constructing resilient multicore systems. In *Proceedings of the 41st annual IEEE/ACM International Symposium on Microarchitecture*, MICRO 41, pages 141–151, Washington, DC, USA, 2008. IEEE Computer Society.

[49] M. R. Guthaus, J. S. Ringenberg, D. Ernst, T. M. Austin, T. Mudge, and R. B. Brown. MiBench: A free, commercially representative embedded benchmark suite. In *Proceedings of the Workload Characterization, 2001. WWC-4. 2001 IEEE International Workshop*, pages 3–14, Washington, DC, USA, 2001. IEEE Computer Society.

[50] G. Halfacree. Gigabyte leak points to Trinity-based Athlon X4. bit-tech news article, September 2012.

[51] R. Hameed, W. Qadeer, M. Wachs, O. Azizi, A. Solomatnikov, B. C. Lee, S. Richardson, C. Kozyrakis, and M. Horowitz. Understanding sources of inefficiency in general-purpose chips. In *Proceedings of the 37th annual international symposium on Computer architecture*, ISCA '10, pages 37–47, New York, NY, USA, 2010. ACM.

[52] A. Hansson, K. Goossens, and A. Rdulescu. Avoiding message-dependent deadlock in network-based systems on chip. *VLSI Design*, 2007:10, 2007.

[53] M. Hill and M. Marty. Amdahl's law in the multicore era. *Computer*, 41(7):33 –38, July 2008.

[54] M. Horowitz. VLSI scaling for architects. vlsiweb.stanford.edu/papers/VLSIScaling.pdf, 2000.

[55] J. Huang, A. Raman, T. B. Jablin, Y. Zhang, T.-H. Hung, and D. I. August. Decoupled software pipelining creates parallelization opportunities. In *Proceedings of the 8th annual IEEE/ACM international symposium on Code generation and optimization*, CGO '10, pages 121–130, New York, NY, USA, 2010. ACM.

[56] INMOS Limited. Transputer instruction set: a compiler writer's guide. www.transputer.net/iset/pdf/tis-acwg.pdf, 1988.

[57] Intel Corporation. Intel 64 and IA-32 architectures optimization reference manual. http://www.intel.com/Assets/en_US/PDF/manual/248966.pdf, June 2011.

[58] Intel Corporation. Intel architecture instruction set extensions programming reference. http://software.intel.com/sites/default/files/319433-014.pdf, August 2012.

[59] E. Ipek, M. Kirman, N. Kirman, and J. F. Martinez. Core Fusion: accommodating software diversity in chip multiprocessors. In *Proceedings of the 34th annual International Symposium on Computer Architecture*, ISCA '07, pages 186–197, New York, NY, USA, 2007. ACM.

[60] J. Johnston and T. Fitzsimmons. The newlib homepage. http://sourceware.org/newlib/, 2011.

[61] A. B. Kahng, B. Li, L.-S. Peh, and K. Samadi. ORION 2.0: a fast and accurate NoC power and area model for early-stage design space exploration. In *Proceedings of the Conference on Design, Automation and Test in Europe*, DATE '09, pages 423–428, 3001 Leuven, Belgium, Belgium, 2009. European Design and Automation Association.

[62] M. Kamruzzaman, S. Swanson, and D. M. Tullsen. Inter-core prefetching for multicore processors using migrating helper threads. In *Proceedings of the sixteenth international conference on Architectural support for programming languages and operating systems*, ASPLOS '11, pages 393–404, New York, NY, USA, 2011. ACM.

[63] P. Kapur, G. Chandra, and K. C. Saraswat. Power estimation in global interconnects and its reduction using a novel repeater optimization methodology. In *Proceedings of the 39th annual Design Automation Conference*, DAC '02, pages 461–466, New York, NY, USA, 2002. ACM.

[64] U. Karpuzcu, B. Greskamp, and J. Torrellas. The BubbleWrap many-core: Popping cores for sequential acceleration. In *Microarchitecture, 2009. MICRO-42. 42nd Annual IEEE/ACM International Symposium on*, pages 447–458, 2009.

[65] S. Keckler, W. Dally, B. Khailany, M. Garland, and D. Glasco. GPUs and the future of parallel computing. *Micro, IEEE*, 31(5):7–17, 2011.

[66] J. H. Kelm, D. R. Johnson, M. R. Johnson, N. C. Crago, W. Tuohy, A. Mahesri, S. S. Lumetta, M. I. Frank, and S. J. Patel. Rigel: An architecture and scalable programming interface for a 1000-core accelerator. In *ISCA '09: Proceedings of the 36th annual international symposium on Computer architecture*, pages 140–151, New York, NY, USA, 2009. ACM.

[67] Khronos Group. OpenCL – the open standard for parallel programming of heterogeneous systems. http://www.khronos.org/opencl/.

[68] Khubaib, M. A. Suleman, M. Hashemi, C. Wilkerson, and Y. N. Patt. MorphCore: an energy-efficient microarchitecture for high performance ILP and high throughput TLP. In *Proceedings of the 45th International Symposium on Microarchitecture*, 2012.

[69] J. S. Kim, M. B. Taylor, J. Miller, and D. Wentzlaff. Energy characterization of a tiled architecture processor with on-chip networks. In *Proceedings of the 2003 international symposium on Low power electronics and design*, ISLPED '03, pages 424–427, New York, NY, USA, 2003. ACM.

[70] J. Kin, M. Gupta, and W. H. Mangione-Smith. The filter cache: an energy efficient memory structure. In *Proceedings of the 30th annual ACM/IEEE international symposium on Microarchitecture*, MICRO 30, pages 184–193, Washington, DC, USA, 1997. IEEE Computer Society.

[71] P. Kongetira, K. Aingaran, and K. Olukotun. Niagara: a 32-way multithreaded Sparc processor. *Micro, IEEE*, 25(2):21 – 29, March-April 2005.

[72] R. Krashinsky, C. Batten, M. Hampton, S. Gerding, B. Pharris, J. Casper, and K. Asanovic. The vector-thread architecture. In *Proceedings of the 31st annual international symposium on Computer architecture*, ISCA '04, pages 52–63, Washington, DC, USA, 2004. IEEE Computer Society.

[73] I. Kuon and J. Rose. Measuring the gap between FPGAs and ASICs. In *Proceedings of the 2006 ACM/SIGDA 14th international symposium on Field programmable gate arrays*, FPGA '06, pages 21–30, New York, NY, USA, 2006. ACM.

[74] T. Lanier. Exploring the design of the Cortex-A15 processor. www.arm.com/files/pdf/at-exploring_the_design_of_the_cortex-a15.pdf, May 2013.

[75] C. Lattner. The LLVM compiler infrastructure. http://llvm.org/.

[76] S. Lavington. *A History of Manchester Computers*. British Computer Society, 1998.

[77] L. H. Lee, B. Moyer, and J. Arends. Low-cost embedded program loop caching - revisited. Technical report, University of Michigan, 1999.

[78] Y. Lee, R. Krashinsky, V. Grover, S. W. Keckler, and K. Asanovic. Convergence and scalarization for data-parallel architectures. In *Code Generation and Optimization, Proceedings of the 2013 International Symposium on*, 2013.

[79] S. Li, J. H. Ahn, R. D. Strong, J. B. Brockman, D. M. Tullsen, and N. P. Jouppi. McPAT: an integrated power, area, and timing modeling framework for multicore and manycore architectures. In *Proceedings of the 42nd Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO 42, pages 469–480, New York, NY, USA, 2009. ACM.

[80] M. Luff. Communication for programmability and performance on multi-core processors. Technical report, University of Cambridge, April 2013.

[81] K. Mai, R. Ho, E. Alon, D. Liu, Y. Kim, D. Patil, and M. Horowitz. Architecture and circuit techniques for a 1.1-GHz 16-kB reconfigurable memory in 0.18-μm CMOS. *IEEE Journal of Solid-State Circuits*, 40:261–275, January 2005.

[82] K. Mai, T. Paaske, N. Jayasena, R. Ho, W. J. Dally, and M. Horowitz. Smart Memories: a modular reconfigurable architecture. In *Proceedings of the 27th annual international symposium on Computer architecture*, ISCA '00, pages 161–171, New York, NY, USA, 2000. ACM.

[83] J. Mak. Facilitating program parallelisation: a profiling-based approach. Technical report, University of Cambridge, March 2011.

[84] R. Manohar and K. M. Chandy. Δ-Dataflow networks for event stream processing. In *Proceedings of the International Conference on Parallel and Distributed Computing and Systems*, November 2004.

[85] M. Mantor and M. Houston. AMD Graphics Core Next. amddevcentral.com/afds/assets/presentations/2620_final.pdf, June 2011.

[86] D. May. The XMOS architecture and XS1 chips. *Micro, IEEE*, 32(6):28–37, 2012.

[87] J. Meng, J. W. Sheaffer, and K. Skadron. Robust SIMD: Dynamically adapted SIMD width and multi-threading depth. In *Proceedings of the IEEE International Parallel & Distributed Processing Symposium (IPDPS)*, May 2012.

[88] M. Mishra and S. Goldstein. Virtualization on the Tartan reconfigurable architecture. In *Field Programmable Logic and Applications, 2007. FPL 2007. International Conference on*, pages 323–330, August 2007.

[89] Monsoon Solutions, Inc. Power monitor. http://www.msoon.com/LabEquipment/PowerMonitor/.

[90] Motorola. *Motorola M68000 Family Programmers Reference Manual*, 1 edition, 1992.

[91] J. Mukundan, S. Ghose, R. Karmazin, E. Ípek, and J. F. Martínez. Overcoming single-thread performance hurdles in the Core Fusion reconfigurable multicore architecture. In *Proceedings of the 26th ACM international conference on Supercomputing*, ICS '12, pages 101–110, New York, NY, USA, 2012. ACM.

[92] N. Muralimanohar, R. Balasubramonian, and N. P. Jouppi. CACTI 6.0: A tool to model large caches, April 2009.

[93] T. Nagatsuka, Y. Sakaguchi, T. Matsumura, and K. Kise. CoreSymphony: an efficient reconfigurable multi-core architecture. *SIGARCH Comput. Archit. News*, 39(4):32–37, Dec. 2011.

[94] W. Najjar, W. Bohm, B. Draper, J. Hammes, R. Rinker, J. Beveridge, M. Chawathe, and C. Ross. High-level language abstraction for reconfigurable computing. *Computer*, 36(8):63 – 69, August 2003.

[95] J. Nickolls, I. Buck, M. Garland, and K. Skadron. Scalable parallel programming with CUDA. *Queue*, 6(2):40–53, Mar. 2008.

[96] J. Oliver, R. Rao, M. Brown, J. Mankin, D. Franklin, F. T. Chong, and V. Akella. Tile size selection for low-power tile-based architectures. In *Proceedings of the 3rd conference on Computing frontiers*, CF '06, pages 83–94, New York, NY, USA, 2006. ACM.

[97] OMNeT++ Community. OMNeT homepage. http://www.omnetpp.org/.

[98] Open SystemC Initiative. SystemC homepage. http://www.systemc.org/home/.

[99] G. Ottoni, R. Rangan, A. Stoler, and D. August. Automatic thread extraction with decoupled software pipelining. In *Microarchitecture, 2005. MICRO-38. Proceedings. 38th Annual IEEE/ACM International Symposium on*, page 12, Nov. 2005.

[100] R. Panda, A. Wood, N. McVicar, C. Ebeling, and S. Hauck. Extending course-grained reconfigurable arrays with multi-kernel dataflow. In *The Second Workshop on the Intersections of Computer Architecture and Reconfigurable Logic (CARL 2012)*, June 2012.

[101] H. Park, Y. Park, and S. Mahlke. Polymorphic pipeline array: a flexible multicore accelerator with virtualized execution for mobile multimedia applications. In *Proceedings of the 42nd Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO 42, pages 370–380, New York, NY, USA, 2009. ACM.

[102] J. Park, J. Balfour, and W. J. Dally. Maximizing the filter rate of L0 compiler-managed instruction stores by pinning. Technical Report 126, Stanford University, 2009.

[103] Y. Park, H. Park, and S. Mahlke. CGRA express: accelerating execution using dynamic operation fusion. In *Proceedings of the 2009 international conference on Compilers, architecture, and synthesis for embedded systems*, CASES '09, pages 271–280, New York, NY, USA, 2009. ACM.

[104] D. Pham, J. Holt, and S. Deshpande. Embedded multicore systems: Design challenges and opportunities. In *Multiprocessor System-on-Chip*, pages 197–222. Springer, 2011.

[105] M. D. Powell, A. Biswas, S. Gupta, and S. S. Mukherjee. Architectural core salvaging in a multi-core processor for hard-error tolerance. In *Proceedings of the 36th annual international symposium on Computer architecture*, ISCA '09, pages 93–104, New York, NY, USA, 2009. ACM.

[106] M. Pricopi and T. Mitra. Bahurupi: A polymorphic heterogeneous multi-core architecture. *ACM Trans. Archit. Code Optim.*, 8(4):22:1–22:21, Jan. 2012.

[107] A. Raghavan, Y. Luo, A. Chandawalla, M. Papaefthymiou, K. P. Pipe, T. F. Wenisch, and M. M. K. Martin. Computational sprinting. In *Proceedings of the 18th Symposium on High Performance Computer Architecture (HPCA)*, February 2012.

[108] A. D. Reid, K. Flautner, E. Grimley-Evans, and Y. Lin. SoC-C: efficient programming abstractions for heterogeneous multicore systems on chip. In *Proceedings of the 2008 international conference on Compilers, architectures and synthesis for embedded systems*, CASES '08, pages 95–104, New York, NY, USA, 2008. ACM.

[109] R. Rodrigues, A. Annamalai, I. Koren, S. Kundu, and O. Khan. Performance per Watt benefits of dynamic core morphing in asymmetric multicores. In *Parallel Architectures and Compilation Techniques (PACT), 2011 International Conference on*, pages 121 –130, Oct. 2011.

[110] P. Rogers. The programmer's guide to the APU galaxy. http://developer.amd.com/afds/assets/keynotes/Phil%20Rogers%20Keynote-FINAL.pdf, 2011.

[111] R. Ronen, A. Mendelson, K. Lai, S.-L. Lu, F. Pollack, and J. Shen. Coming challenges in microarchitecture and architecture. *Proceedings of the IEEE*, 89(3):325–340, 2001.

[112] A. Ros and S. Kaxiras. Complexity-effective multicore coherence. In *Proceedings of the 21st international conference on Parallel architectures and compilation techniques*, PACT '12, pages 241–252, New York, NY, USA, 2012. ACM.

[113] D. Sanchez, R. M. Yoo, and C. Kozyrakis. Flexible architectural support for fine-grain scheduling. *SIGARCH Comput. Archit. News*, 38(1):311–322, Mar. 2010.

[114] K. Sankaralingam, R. Nagarajan, R. McDonald, R. Desikan, S. Drolia, M. S. Govindan, P. Gratz, D. Gulati, H. Hanson, C. Kim, H. Liu, N. Ranganathan, S. Sethumadhavan, S. Sharif, P. Shivakumar, S. W. Keckler, and D. Burger. Distributed microarchitectural protocols in the TRIPS prototype processor. In *Proceedings of the 39th Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO 39, pages 480–491, Washington, DC, USA, 2006. IEEE Computer Society.

[115] G. Sârbu. Power modelling and validation. Master's thesis, University of Cambridge, www.cl.cam.ac.uk/∼gs448/MPhil_dissertation_gs448.pdf, June 2012.

[116] Semico Research Corp. Complex SoC silicon and software design costs are skyrocketing. http://www.design-reuse.com/news/27117, August 2011.

[117] S. Shapiro and M. Wilk. An analysis of variance test for normality (complete samples). *Biometrika*, 52(3/4):591–611, 1965.

[118] D. She, Y. He, and H. Corporaal. Energy efficient special instruction support in an embedded processor with compact ISA. In *Proceedings of the 2012 international conference on Compilers, architectures and synthesis for embedded systems*, CASES '12, pages 131–140, New York, NY, USA, 2012. ACM.

[119] D. G. Shea, R. C. Booth, D. H. Brown, M. E. Giampapa, G. R. Irwin, T. T. Murakami, F. T. Tong, P. R. Varker, W. W. Wilcke, and D. J. Zukowski. Monitoring and simulation of processing strategies for large knowledge bases on the IBM Victor multiprocessor. In J. John A. Board, editor, *Transputer Research and Applications 2*, pages 11–25. IOS, 1990.

[120] A. L. Shimpi. ASUS Transformer Pad Infinity (TF700T) review. http://www.anandtech.com/show/6036/3, June 2012.

[121] N. Slingerland and A. J. Smith. Performance analysis of instruction set architecture extensions for multimedia. In *the 3rd Workshop on Media and Stream Processors*, pages 204–217, 2001.

[122] SoC Consortium. ARM processor architecture. Lecture notes: http://access.ee.ntu.edu.tw/course/SoC_Lab_981/lecture.htm, May 2013.

[123] SPMT Consortium. SPMT homepage. www.spmt.org.

[124] M. B. Taylor, J. Kim, J. Miller, D. Wentzlaff, F. Ghodrat, B. Greenwald, H. Hoffman, P. Johnson, J.-W. Lee, W. Lee, A. Ma, A. Saraf, M. Seneski, N. Shnidman, V. Strumpen, M. Frank, S. Amarasinghe, and A. Agarwal. The Raw microprocessor: A computational fabric for software circuits and general-purpose programs. *IEEE Micro*, 22:25–35, March 2002.

[125] M. B. Taylor, W. Lee, S. P. Amarasinghe, and A. Agarwal. Scalar operand networks. *IEEE Transactions on Parallel Distributed Systems*, 16(2):145–162, 2005.

[126] J. Teich, A. Weichslgartner, B. Oechslein, and W. Schroder-Preikschat. Invasive computing - concepts and overheads. In *Specification and Design Languages (FDL), 2012 Forum on*, pages 217 –224, Sept. 2012.

[127] Tilera Corporation. Tilera processors webpage. http://www.tilera.com/products/processors.php.

[128] D. Truong, W. Cheng, T. Mohsenin, Z. Yu, A. Jacobson, G. Landge, M. Meeuwsen, C. Watnik, A. Tran, Z. Xiao, E. Work, J. Webb, P. Mejia, and B. Baas. A 167-processor computational platform in 65 nm CMOS. *Solid-State Circuits, IEEE Journal of*, 44(4):1130 –1144, April 2009.

[129] University of Edinburgh School of Informatics. HASE userguide. www.icsa.informatics.ed.ac.uk/research/groups/hase/manuals/hasepp/hasepp.html, May 2010.

[130] G. Venkatesh, J. Sampson, N. Goulding, S. Garcia, V. Bryksin, J. Lugo-Martinez, S. Swanson, and M. B. Taylor. Conservation cores: reducing the energy of mature computations. In *ASPLOS '10: Proceedings of the fifteenth edition of ASPLOS on Architectural support for programming languages and operating systems*, pages 205–218, New York, NY, USA, 2010. ACM.

[131] X. Vera, B. Lisper, and J. Xue. Data cache locking for tight timing calculations. *ACM Trans. Embed. Comput. Syst.*, 7(1):4:1–4:38, Dec. 2007.

[132] D. Wentzlaff, P. Griffin, H. Hoffmann, L. Bao, B. Edwards, C. Ramey, M. Mattina, C.-C. Miao, J. Brown, and A. Agarwal. On-chip interconnection architecture of the Tile Processor. *Micro, IEEE*, 27(5):15–31, 2007.

[133] N. Weste and D. Harris. *CMOS VLSI Design*. Addison-Wesley, 2010.

[134] M. Woh, S. Seo, S. Mahlke, T. Mudge, C. Chakrabarti, and K. Flautner. AnySP: anytime anywhere anyway signal processing. In *Proceedings of the 36th annual international symposium on Computer architecture*, ISCA '09, pages 128–139, New York, NY, USA, 2009. ACM.

[135] Xilinx. Xilinx 7 series FPGAs overview. www.xilinx.com/support/documentation/data_sheets/ds180_7Series_Overview.pdf, November 2012.

[136] H. Zhong, M. Mehrara, S. Lieberman, and S. Mahlke. Uncovering hidden loop level parallelism in sequential applications. In *In Proc. of the 14th International Symposium on High-Performance Computer Architecture*, 2008.

APPENDIX A

INSTRUCTION SET ARCHITECTURE

These tables and reference pages give an overview of the Loki instruction set architecture. Much of this information can be found elsewhere in the dissertation, and is summarised here for convenience.

A.1 Datapath

This section describes ways in which a pipeline can store, retrieve and communicate data.

A.1.1 Registers

Each core in the Loki architecture has a 32-entry register file accessible through two read ports and one write port. All registers are accessible through the instruction set. Some registers are mapped to network buffers: read operations at these positions are destructive, removing the contents, and blocking, so the core will stall if data has not yet arrived. The uses of registers are summarised in Table A.1.

A.1.2 Predicates

The Loki architcture supports predicated execution of all instructions to make simple control flow more efficient and allow SIMD execution in the presence of slightly-divergent execution paths.

Many instructions have a `.p` variant, which stores a one-bit value in the core's predicate register at the end of the execute pipeline stage. This value may be the least significant bit of the result of bitwise operations, the result itself in the case of Boolean operations, or the carry/borrow flag for arithmetic operations.

Register	Description
0	Constant 0
1	Memory address of first instruction in current instruction packet
2-7	Input network buffers
8-31	General-purpose registers

Table A.1: Register uses.

Encoding	Meaning
00	Execute if predicate = 1
01	Execute if predicate = 0
10	Always execute
11	End of packet (always execute)

Table A.2: Predicate encodings.

Channel index	Hardware structure
0	Instruction buffer
1	Instruction packet cache
2-7	Registers 2-7

Table A.3: Hardware components corresponding to each input channel of a Loki core.

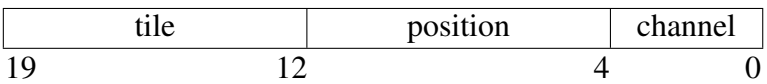
All instructions may be executed conditionally, based on the value of the predicate register. Two bits are reserved in the instruction encoding to determine when an instruction should execute. An `ifp?` prefix causes an instruction to execute only if the predicate is true, and `if!p?` instructions only execute when the predicate is false. The remaining two options are *always execute* (default), and *end of packet* (denoted with an `.eop` marker in assembly). This information is summarised in Table A.2.

A.1.3 Channels

Each core has eight input channels and sixteen output channels. Each input channel is associated with a separate buffer, as described in Table A.3, so that data from different sources can be accessed in a different order to the one in which they arrived. All output channels are multiplexed onto a single output buffer, and their uses are summarised in Table A.4. Network communication is blocking: the core will stall if data has not yet arrived on an input channel, or if the output buffer is full.

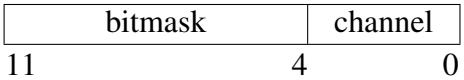
Network addresses are encoded as integers, allowing them to be manipulated by the ALU before they are stored in the channel map table, or shared between components. It is possible to specify point-to-point addresses to any component on the chip, and multicast addresses within a tile.

A point-to-point address is encoded as:



tile is the index of the tile on the chip; *position* is the component's position within the tile – cores range from 0 to 7 and memories range from 8 to 15; *channel* is the component's channel to be accessed – cores have 8 input channels and memories have 16.

Multicast addresses are encoded as:



Channel index	Description
0	Implicitly used by *fetch* and *fill* instructions
1-14	General-purpose channels
15	Null channel mapping: used in situations where an instruction's result should not be sent onto the network

Table A.4: Use cases for each output channel of a Loki core.

bitmask is an 8-bit entry with one bit for each core in the local tile. The core at position 0 is represented by the least significant bit. Data sent to this address is to be sent to all cores which have their bits set in the bitmask. To simplify the encoding and reduce the amount of information required, a restriction is made that the data must be sent to the same input channel of all target cores. This restriction did not prove to be a limitation in any of the experiments which used multicast.

A.2 Instruction formats

Loki's simple instruction formats make decoding inexpensive: all fields except the immediate remain in the same positions across all formats. In the formats below, *p* represents the predicate bits, allowing execution to be conditional on the value of the predicate register, and marking the ends of instruction packets. *fn* is the ALU function.

FF (fetch) format:

p	opcode	immediate
2	7	23

0R (zero registers) format:

p	opcode	xxxxx	channel	immediate
2	7	5	4	14

0Rnc (zero registers, no channel) format:

p	opcode	xxxxx	immediate
2	7	9	14

1R (one register) format:

p	opcode	reg1	channel	immediate
2	7	5	4	14

1Rnc (one register, no channel) format:

p	opcode	reg1	xx	immediate
2	7	5	2	16

2R (two registers) format:

p	opcode	reg1	channel	reg2	immediate
2	7	5	4	5	9

2Rnc (two registers, no channel) format:

p	opcode	reg1	xxxx	reg2	immediate
2	7	5	4	5	9

2Rs (two registers with shift amount) format:

p	opcode	reg1	channel	reg2	xxxx	immediate
2	7	5	4	5	4	5

3R (three registers) format:

p	opcode	reg1	channel	reg2	reg3	fn
2	7	5	4	5	5	4

A.3 Instruction summary

Table A.5 summarises all instructions in the Loki ISA. The *.p* field indicates whether the instruction has a variant which writes to the predicate register, the mnemonic of which is `mnemonic.p`.

Table A.5: Instruction set summary.

Mnemonic	.p	Name	Format
ADDU	✓	Addition	3R
ADDUI	✓	Addition with immediate	2R
AND	✓	Bitwise AND	3R
ANDI	✓	Bitwise AND with immediate	2R
CLZ	✗	Count leading zeros	2R
FETCH	✗	Fetch	FF
FETCHPST	✗	Fetch persistent	1Rnc
FETCHPSTR	✗	Fetch persistent relative	FF
FETCHR	✗	Fetch relative	FF
FILL	✗	Fill	1Rnc
FILLR	✗	Fill relative	FF
IBJMP	✗	In buffer jump	0R
IRD	✗	Indirect read	2R
IWR	✗	Indrect write	2R
LDBU	✗	Load byte	1R
LDHWU	✗	Load halfword	1R
LDW	✗	Load word	1R
LLI	✗	Load lower immediate	1Rnc
LUI	✗	Load upper immediate	1Rnc
MULHW	✗	Multiply high word	3R
MULHWU	✗	Multiply high word (unsigned)	3R
MULLW	✗	Multiply low word	3R

Continued on next page

Table A.5 – *Continued from previous page*

Mnemonic	.p	Name	Format
NOR	✓	Bitwise NOR	3R
NORI	✓	Bitwise NOR with immediate	2R
OR	✓	Bitwise OR	3R
ORI	✓	Bitwise OR with immediate	2R
PSEL	✗	Select on predicate bit	3R
PSEL.FETCH	✗	Fetch with select address on predicate	2Rnc
RMTEXECUTE	✗	Send instructions to another processor	0R
RMTNXIPK	✗	Send next instruction packet command	0R
SCRATCHRD	✗	Scratchpad read	2R
SCRATCHRDI	✗	Scratchpad read immediate	1R
SCRATCHWR	✗	Scratchpad write	2R
SCRATCHWRI	✗	Scratchpad write immediate	1R
SELCH	✗	Select a channel-end with data	1Rnc
SETCHMAP	✗	Set channel map entry	2Rnc
SETCHMAPI	✗	Set channel map entry immediate	1Rnc
SETEQ	✓	Set if equal	3R
SETEQI	✓	Set if equal with immediate	2R
SETFETCHCH	✗	Set fetch channel	0R
SETGTE	✓	Set if greater than or equal	3R
SETGTEI	✓	Set if greater than or equal to immediate	2R
SETGTEU	✓	Set if greater than or equal (unsigned)	3R
SETGTEUI	✓	Set if greater than or equal to immediate (unsigned)	2R
SETLT	✓	Set if less than	3R
SETLTI	✓	Set if less than immediate	2R
SETLTU	✓	Set if less than (unsigned)	3R
SETLTUI	✓	Set if less than immediate (unsigned)	2R
SETNE	✓	Set if not equal	3R
SETNEI	✓	Set if not equal with immediate	2R
SLL	✗	Shift left logical	3R
SLLI	✗	Shift left logical by immediate	2Rs
SRA	✗	Shift right arithmetic	3R
SRAI	✗	Shift right arithmetic by immediate	2Rs
SRL	✓	Shift right logical	3R
SRLI	✓	Shift right logical by immediate	2Rs
STB	✗	Store byte	2R
STHW	✗	Store halfword	2R
STW	✗	Store word	2R
SUBU	✓	Subtraction	3R
SYSCALL	✗	System Call	0Rnc
TSTCH	✓	Test input channel-end	2R
TSTCHI	✓	Test input channel-end immediate	1R
WOCHE	✗	Wait until output channel is empty	0R
XOR	✓	Bitwise XOR	3R
XORI	✓	Bitwise XOR with immediate	2R

Class	Mnemonic	Description
Comparison	seteq	Set if equal
	setne	Set if not equal
	setlt_s	Set if less than (signed)
	setlt	Set if less than (unsigned)
	setgte_s	Set if greater than or equal (signed)
	setgte	Set if greater than or equal (unsigned)
Shift	sll	Shift left logical
	srl	Shift right logical
	sra	Shift right arithmetic
Arithmetic	add	Addition
	sub	Subtraction
Bitwise logic	nor	Negated OR
	and	AND
	or	OR
	xor	Exclusive OR
	nand	Negated AND
	clr	Clear; AND with negated second operand
	orc	OR complement; OR with negated second operand
Miscellaneous	clz	Count leading zeroes
	lui	Load upper immediate

Table A.6: ALU functions

A.4 Instruction reference

A.4.1 ALU

The ALU is capable of the 20 functions presented in Table A.6. The function is provided directly by instructions which use the 3 register format, and the decoder provides the function for all other ALU operations.

Most ALU operations have the optional ability to write to the predicate register. The comparison operations write their 1-bit result, the bitwise and shift operations write the least significant bits of their results, and the arithmetic operations use the predicate register as an integer overflow flag.

A.4.2 Data

This section describes instructions which manipulate local data without using the ALU.

lli (*load lower immediate*) is the partner for the *lui* ALU function and stores a 16 bit immediate in a register.

The multiplier takes 32-bit operands and produces one word of the result in two clock cycles. There are three multiplication instructions: *mullw* (low word), *mulhw* (high word) and *mulhwu* (high word, where operands are unsigned).

Loki supports indirect access to its registers and input buffers through the *indirect read* and *write* instructions. These allow the contents of one register to be used as the index to be accessed. Indirect reads take two clock cycles because the two reads must be performed sequentially. These instructions are particularly useful when combined with *selch* (Section A.4.5) as they allow data to be read from whichever input buffer currently has data.

Similar instructions are provided to access the scratchpad memory: *scratchrd* and *scratchwr*. These access indices held in a register. Additional forms of the two instructions which take an immediate argument are also provided.

psel (*predicated select*) takes two register operands and selects one based on the value of the predicate register. In the case where *psel* may perform a destructive read from an input buffer, a one-cycle bubble is inserted if the previous instruction writes to the predicate register. This allows time for the predicate to be computed and ensures that the data is only consumed if it is required.

A.4.3 Instruction fetch

Loki offers multiple ways of managing the contents of each core's instruction cache and buffer.

The *fetch* instruction requests an instruction packet if it is not already cached locally. The packet is queued up to execute as soon as the current packet finishes, which means it can be prefetched well in advance to hide memory latency. There can be at most one *fetch* in progress at any one time, to avoid the need for deinterleaving instructions from different packets. There is a one-cycle branch delay slot when there is a hit in the L0 instruction cache: the instruction packet's memory address is computed (or read from a register) in the decode pipeline stage, and the L0 lookup is performed in the following cycle in the fetch stage. There are variants of the instruction which use absolute addresses, relative addresses, and select between two addresses based on the value in the predicate register.

fill requests an instruction packet if it is not already cached locally, but does not execute it. This is useful if there are instruction packets which are known statically to be good to store in the cache. Once the cache has been filled with useful instructions, the contents can effectively be locked by directing all subsequent instructions to the instruction buffer instead.

Persistent instruction packets are instruction packets which execute repeatedly until either a *next instruction packet* command (Section A.4.6) is received over the network, or a new packet is fetched. An instruction packet is made persistent by using the *fetch persistent* instruction in the place of an ordinary *fetch*. Each iteration of the instruction packet can be issued immediately after the previous one finishes; there is no bubble in the pipeline. When the core drops out of persistent mode, the pipeline is flushed, and any remaining instructions in the current iteration are cancelled. Persistent instruction packets are useful for implementing very tight loops, since the branch instruction is not needed. Predicated execution can be used to add simple control flow to a persistent packet. Section 6.4.4 explored adding two variants of *fetch persistent* which continued execution until the predicate register held a certain value.

Originally, all *fetch* requests were sent over the default logical channel 0. The cache pinning experiments in Section 5.1.3 found that providing multiple channels to the instruction memory would be useful as it would allow the core to switch between its instruction inputs more quickly to control energy consumption. *setfetchch* (*set fetch channel*) was proposed which takes only a logical network address as an argument and sets it as the default channel used by *fetch* instructions.

The *in-buffer jump* instruction means that if the contents of the cache are known statically, the core may jump around within its cache without the need for computing a memory address and checking all cache tags. This instruction has the advantage that it completes one pipeline stage earlier than the others, as no address computation needs to be performed. This means that there is no branch delay slot, and the next instruction can be issued in the following clock cycle.

A.4.4 Memory

Loki supports loading and storing bytes, halfwords and words. Since memory banks are accessed over the network, each operation must specify a network channel over which memory can be reached, and the memory bank must know where to send any results back to (Section 3.4).

Sending a load request to memory and making use of the result are two separate operations, and can be separated by any number of intermediate instructions to hide memory latency. When there is no contention at the memory banks, latency is two cycles, meaning that two intermediate instructions are required to keep the pipeline full. Care must be taken if another *load* is used as an intermediate instruction: it is possible for the two memory operations to access different banks and complete out of order. This can be avoided by allocating multiple memory channels and switching between them when necessary.

Store operations take two clock cycles to complete as they must generate a memory address as well as collect the data to be stored.

A.4.5 Network

Since the network is such a major component of Loki's design, there exist a number of instructions to manage it.

setchmap (*set channel map*) associates a physical network address with a logical address in the channel map table. Physical addresses are of the form shown in Section A.1.3 and logical addresses are supplied either from a register or as an immediate value.

tstch (*test channel*) returns a Boolean result telling whether the selected input buffer contains any data. This allows the core to perform more work if data has not yet arrived, instead of performing a blocking read operation.

selch (*select channel*) returns the register mapping of any input buffer which holds data. If all buffers are empty, the pipeline stalls until data arrives. This is useful in situations where data can arrive from many different sources, but can be processed in any order.

woche (*wait until output channel is empty*) stalls the pipeline until all data sent on a particular channel has reached its destination. In the case of local communication, data is only injected into the network when it has been allocated a path to its destination, so the instruction simply completes when all data has left the core's output buffer. For communication to a distant tile, a credit counter must also be checked. The instruction guarantees that it is safe to allow another core to begin sending data to the same destination – all data has arrived so there is no possibility of interleaving.

A.4.6 Remote execution

Loki cores have the ability to send instructions over the network to other cores; a process known as remote execution. This is useful when a core wants to offload a small amount of work, or

when it is cheaper to move the instructions to the data than it would be to bring the data to the instructions.

Following a *rmtexecute* instruction, all consecutive instructions marked with an `ifp?` predicate will be sent to the specified destination, and will not be executed locally. All instructions sent remotely have the `ifp?` predicate replaced with a `.eop` end of packet marker, so that the remote core switches to an alternate source of instructions as soon as they begin to arrive. For this reason, the *instruction fetch* must be the final instruction sent to the remote core – the incoming individual instructions are indistinguishable from the packet being fetched as far as the core is concerned.

The execution of another core can be stopped using *rmtnxipk* (*remote next instruction packet*). This instruction sends a special command to the remote core, which when received, immediately causes a pipeline flush and a switch to the next instruction packet (if any). This is particularly useful to break cores out of infinite loops and to reactivate them if they are stalled waiting for data which will never arrive.

A.4.7 Other

Loki has support for a *system call* instruction to request service from an operating system. The only argument is an immediate which selects the service the operating system should provide.