Technical Report

UCAM-CL-TR-832
ISSN 1476-2986

Number 832

UNIVERSITY OF CAMBRIDGE

Computer Laboratory

Communication centric, multi-core, fine-grained processor architecture

Gregory A. Chadwick

April 2013

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 September 2012 by the author for the degree of Doctor of Philosophy to the University of Cambridge, Fitzwilliam College.

ISSN 1476-2986

Abstract

With multi-core architectures now firmly entrenched in many application areas both computer architects and programmers now face new challenges. Computer architects must increase core count to increase explicit parallelism available to the programmer in order to provide better performance whilst leaving the programming model presented tractable. The programmer must find ways to exploit this explicit parallelism provided that scale well with increasing core and thread availability.

A fine-grained computation model allows the programmer to expose a large amount of explicit parallelism and the greater the level of parallelism exposed the better increasing core counts can be utilised. However a fine-grained approach implies many interworking threads and the overhead of synchronising and scheduling these threads can eradicate any scalability advantages a fine-grained program may have.

Communication is also a key issue in multi-core architecture. Wires do not scale as well as gates, making communication relatively more expensive compared to computation so optimising communication between cores on chip becomes important.

This dissertation presents an architecture designed to enable scalable fine-grained computation that is communication aware (allowing a programmer to optimise for communication). By combining a tagged memory, where each word is augmented with a presence bit signifying whether or not data is present in that word, with a hardware based scheduler, which allows a thread to wait upon a word becoming present with low overhead. A flexible and scalable architecture well suited to fine-grained computation can be created, one which enables this without needing the introduction of many new architectural features or instructions. Communication is made explicit by enforcing that accesses to a given area of memory will always go to the same cache, removing the need for a cache coherency protocol.

The dissertation begins by reviewing the need for multi-core architecture and discusses the major issues faced in their construction. It moves on to look at fine-grained computation in particular. The proposed architecture, known as Mamba, is then presented in detail with several software techniques suitable for use with it introduced. An FPGA implementation of Mamba is then evaluated against a similar architecture that lacks the extensions Mamba has for assisting in fine-grained computation (namely a memory tagged with presence bits and a hardware scheduler). Microbenchmarks examining the performance of FIFO based communication, MCS locks (an efficient spin-lock implementation based around queues) and barriers demonstrate Mamba's scalability and insensitivity to thread count. A SAT solver implementation demonstrates that these benefits have a real impact on an actual application.

Acknowledgments

Firstly, I would like to thank my Supervisor Simon Moore. Not only for his valuable guidance and advice during my PhD but for his support and teaching from when I first came to know him as an undergraduate. From the time I timidly requested to borrow an DE2 board in the Part 1B ECAD labs for my own self study he always given me the freedom to follow my ideas.

No work is ever done in isolation and my friends and colleagues in the computer architecture group have provided many useful hours of discussion and advice. I must give particular thanks to Arnab Banerjee for his Bluespec implementation of the network I used as well as Theo Markettos for his proof reading of this dissertation.

Finally I would like to thank my family for their support and my friends, for making my time at Cambridge so enjoyable.

Contents

List of Figures

List of Tables

Listings

Chapter 1

Introduction

1.1 Overview

Increasing core counts (rather than increasing transistor counts within a single core) are now driving increasing performance. They also demand innovation within software, available cores must be effectively used in order to gain performance proportional to gain in core count. It is critical that developments in processor architecture enable rather than stifle this innovation in software (massively increased core counts are useless if writing software to take full advantage of them is almost intractable).

In order allow the creation of software for a multi-core machine that can achieve reasonable performance the familiar programming model of a single thread of execution working on a single uniform memory space must be disrupted due to the reality of multiple caches and the need for multiple threads working together to utilise more than one core. Whilst cache coherency enables the illusion of a single flat memory space to remain due regard must be given to the communication generated by a cache coherency protocol to achieve good performance.

Wires do not scale as well as gates, this has led to the need for interconnection networks on chip, to avoid long global wires that span the entire chip and instead have a network using many smaller local wires that have a size relative to the size of a gate. When feature sizes decreases the network can be expanded to contain more nodes but the node to node wire length stays the same size relative to a gate allowing better wire scaling. Communication costs become a critical design point when building software, efficient use of the network is required to achieve the best performance [76].

To achieve good scalability with increasing core count in software a sufficient number of threads, capable of operating in parallel, need to be made available so all cores can be fully utilised. An obvious way to ensure that a piece of software remains scalable as core count increases, rather than being specifically optimised for a certain number of cores, is the use of fine-grained techniques. Here a task is broken down into many small sub-tasks that can potentially be executed in parallel. Provided the number of sub-tasks is more than the number of available cores the software will continue to scale. The more fine-grained the task breakdown is the further this scaling can continue. An obvious flaw in this argument is breaking a task down into finer and finer grained sub-tasks incurs higher and higher overhead in terms of scheduling those sub-tasks and synchronising between them. Fundamentally scaling will be limited by those overheads [31].

An architecture which enables low overhead synchronisation and thread scheduling will enable good scaling with increasing core counts in software that utilises fine-grained techniques. The architecture presented in this dissertation, known as Mamba, aims to do this.

Mamba accomplishes its aims by using a hardware scheduler combined with a tagged memory. Each 64-bit word in memory has an associated presence bit, when this bit is unset the word is empty, otherwise it is full. This forms the basis for thread synchronisation. When a thread accesses a non-present word it waits until it becomes present before continuing. This interacts with the hardware scheduler so a thread that is waiting for a word may be descheduled and only becomes ready again so it may be rescheduled when the word it is waiting for becomes present. Threads in Mamba have a lightweight in memory representation allowing cheap thread creation as well as a number of threads only limited by the size of memory.

Mamba also makes communication explicit. Instead of a complex cache coherency protocol each core is assigned an area of physical address space. A core may only cache locations within its address space, an access to a location outside of its address space must go to the core that owns that area of address space. This ensures there is never any duplication over caches and hence coherency is maintained without an explicit cache coherency protocol. Because of this software can choose precisely when it communicates over the interconnect and control communication costs by arranging data appropriately.

Mamba was created not as a direct replacement for existing architectures but as a way to explore ideas in fine-grained computation. It is designed to achieve scaling up to many thousands of cores (though due to practical limitations the evaluation presented in this dissertation uses a far smaller number). Mamba was created with no particular application area in mind, its primary aim is to demonstrate the effectiveness of its presence bit based synchronization model which is a general and flexible mechanism which could be used in many application areas. An important aspect of Mamba is its performance robustness, once a maximum speed-up has been achieved (where all execution resources are fully utilised) introducing further threads does not significantly degrade performance. This allows the possibility of further scaling if more core were to be introduced.

1.2 Thesis

Given that a fine-grained programming model allows good scaling with increasing core count provided synchronisation and scheduling overhead is kept low. A hardware threading model combined with per-word notification and synchronisation achieved with presence bits is an effective way to enable this.

1.3 Outline

Chapter 2 discusses the background for this work. The trends of silicon integrated circuit technology are reviewed as the driving force behind the need for multi-core architecture. The challenges in building multi-core systems and some solutions are introduced looking at physical connections, interconnection architecture, how an architecture can use communication and how software can use architectural features to build multi-threaded applications. The chapter concludes with a look at some existing multi-core architectures, examining how each has approached the previously discussed challenges.

Chapter 3 discusses fine-grained techniques as a way of enabling scalable software on multi-core architecture. The motivations behind fine-grained techniques are examined and existing fine-grained hardware and software is presented.

Chapter 4 introduces and gives the full detail of the Mamba architecture. Following from the discussion on fine-grained techniques in chapter 3 and the challenges of multi-core design in chapter 2 the motivations behind the design of Mamba are presented. This is followed by a full description of the programming model, the micro architecture of the Mamba core and the arrangement of a full Mamba system.

Chapter 5 examines how to write software for Mamba. Basic primitives using presence bits are introduced and a lock, barrier and FIFO are implemented using them. A SAT solver is implemented as a complete software example.

Chapter 6 evaluates an FPGA implementation of the Mamba architecture. An FPGA implementation of a MIPS64 core using the same memory, cache and interconnection architecture as Mamba but with a pure software scheduler and no presence bit mechanism is used as a comparison system. Microbenchmarks are constructed to evaluate the lock, barrier and FIFO implementations against comparable MIPS64 implementations. The performance of the SAT solver is compared between the MIPS64 and Mamba systems.

Chapter 7 concludes the dissertation, summarising the benefits brought by the Mamba architecture, comparisons are drawn with related work and future directions work on the architecture could take are discussed.

1.4 Published work

An overview of the architecture presented in this dissertation along with a preliminary evaluation performed with lock and FIFO communication benchmarks was published and presented at the 30th International Conference on Computer Design (ICCD) under the title ‘Mamba: A Scalable Communication Centric Multi-Threaded Processor Architecture’ in October 2012. The paper can be found in Appendix A and provides a condensed description of the Mamba architecture without the full detail found in the Mamba architecture chapter.

1.5 Contributions

The design and implementation of a processor architecture with a hardware based scheduler that utilises memory tagged with presence bits to achieve fine-grained synchronisation and notification and allow a number of threads only limited by memory with limited impact on performance

A synchronisation mechanism based upon the MCS lock [73] named the ‘notify chain’, that combined with the presented architecture allows multiple threads to efficiently wait for a particular word to be written

Implementations of a lock, barrier and single producer multiple consumer mechanism based on the notify chain

A single producer single consumer FIFO based upon [58] using presence bits to support efficient waiting on that FIFO by both producer and consumer.

An evaluation of the presented architecture using the lock, barrier and FIFO implementations as well as a SAT solver as an example of a real application.

Chapter 2

Background

From the dawn of the microprocessor in 1971, computer architects have been enjoying the exponential growth in computing power as measured by the number of operations that can be completed in one second. A major part of this growth is owed to the observed exponential trend in transistor density known as Moore's Law. Historically the growth in available transistors has been used to increase the instructions per cycle (IPC) a single processor core is capable of, using increasingly sophisticated architectural methods. This, along with the increase in clock speed available with shrinking transistor geometries, has provided the massive gains in available computing power.

However in recent years we have reached a point where simply increasing the complexity of a single core to increase IPC is becoming increasingly difficult or simply no longer possible. For a given fixed die size the area of a chip reachable in a single clock cycle shrinks with each reduction in feature size, which limits the ability for architects to construct ever more sophisticated ways of increasing IPC in a single core. Further difficulties are encountered in scaling clock speeds ever higher. Designs have hit a power wall, only so much heat can be dissipated which limits the maximum clock speed. An increased clock speed would also lead to an even smaller area of the chip reachable in a single cycle [86, 3].

The solution to the problem is multi-core design. Rather than exploiting instruction level parallelism (ILP), extracted from a serial program by a processor via dynamic methods, we turn to thread level parallelism (TLP), specified explicitly by programmer using a number of threads of execution. Rather than increasing the size of a core to increase ILP exploitation with increased transistor budgets the cores can remain the same size whilst the number is increased to make greater TLP available to the programmer. This brings a new set of challenges to both the architect responsible for the processors design and the programmer responsible for producing the software to run on the processor.

A primary concern for both architect and programmer is communication. A thread must work together with other threads to complete their tasks which necessitates communication amongst them. So an architecture must provide communication mechanisms that a programmer can sensibly use. In a multi-core design communication could be split into two types, local and global. Local communication is communication within a core and given a fixed core size still scales well with decreasing feature size. Global communication is communication between cores where the time (in terms of clock cycles) and power required to communicate over a fixed length does not scale well with decreasing feature size. Communication between two cores that are next to each other may scale well (as the wire distance between them scales with feature size as more cores fit on the same die) but communication between increasingly distant cores becomes increasingly more expensive. This means it is important to optimise for this style of communication. One method to achieve this is to make core   core communication explicit and controllable by the programmer so software can be designed to best optimise for it.

In software inter-thread communication can be split into two styles, implicit and explicit. An implicit style uses an address space shared by a group of threads. To communicate a thread writes into this shared space and the writes are made visible to the other threads that share this space, this can be accomplished via a cache coherency mechanism. In an explicit style specific message passing communication primitives are supplied to the programmer (either directly using available hardware primitives or abstractions supplied by the operating system building on available hardware primitives). Threads then explicitly communicate via messages passed between them.

This chapter discusses multi-core design primarily focused on communication and synchronisation methods. This discussion has been split into four separate sections

Physical connections How cores can be physically connected and the design challenges this brings.

Interconnect architecture How physical connections can be used to create an interconnect over which cores can communicate.

Architectural communication How a multi-core architecture might use an interconnect.

Software communication How software might use architectural features to enable inter-thread communication and synchronisation.

2.1 Physical connections

Historically, physical connections were not of much concern to a processor architect. Whilst the data flow between blocks in a processor was of course important the primary concern was the best utilisation of limited computational resources due to the available transistor budget. Gate delay dominated wire delay so connections between blocks, provided by wires, may have presented a challenge to place and route but had little impact on guiding the design of architecture.

More recently wire delay has come to dominate gate delay. Increasing transistor budgets allows greater and greater computational resources to become available to a processor architect. How blocks should be connected to allow the best use of the available computational resources rather than just how to maximise the use of the available transistors (whilst assuming any connectivity wanted will be available) becomes a primary concern [76].

Ron Ho [86] examined the scaling of on-chip wires in detail. A distinction was made between two different types of wire, local wires which span a fixed number of gates and global wires which span a fixed physical length. With decreasing feature size local wires decrease in length whilst global wires do not. Local wires were predicted to scale well with a 10x penalty vs. gate delay over nine process generations whilst global wires were predicted to scale poorly with a 2000x penalty vs. gate delay over nine process generations. The scaling of wires can be improved with the use of repeaters bringing the scaling trend to a 40x penalty vs. gate delay over nine process generations for global wires and to a 2-3x penalty vs. gate delay over nine process generations for local wires.

An obvious implication of the predicted global wire scaling is that with decreasing feature size the total die area that can be reached in a single clock cycle decreases. As local wires do scale a given processor core of fixed design can continue to be scaled down, adding further features to the core (increasing the area needed and wire lengths required in terms of gates) will become increasingly difficult especially if the increased wire delay encountered is to remain architecturally invisible [3].

A natural solution to this problem is a multi-core design. A core may occupy roughly the same area in terms of gates with each new generation so for a fixed physical die size the number of cores available will increase. The challenge is now how to connect these cores, point to point links between them should scale with the core, but the delay between increasingly distant (as measured by the number of gates or cores between them) cores on a chip will increase with each generation. The design of interconnection networks is discussed later in the chapter.

Another impact of poor global wire scaling whilst not directly related to communication is the distribution of a clock. Building a clock tree that distributes a single clock to an entire chip, whilst maintaining sharp edges and low skew becomes increasingly difficult and uses increasingly larger proportions of a chip's power budget. A solution to this is to have an architecture that doesn't require a globally synchronous clock but instead uses multiple locally synchronous clock domains. This type of design works well with a multi-core architecture with each core running off its own clock with the clock domain cross-over handled by the interconnection network.

2.1.1 Off chip connections

Off chip communication is a major concern, specifically to memory. An optimised on core communication architecture is useless without sufficient memory bandwidth to feed it. Unlike the problems of wire scaling though the so called memory wall has been an issue in computer architecture for many years. With multiple cores all with potentially very different working sets the problem grows worse, ideally each core could have its own memory channel or a channel shared between a small number of cores. Chip packaging constrains the number of pins available, constraining the number of possible off chip memory channels. Recent developments in 3D integration technologies could help combat this. Memory could be stacked directly on top of the cores in the same chip package, this could both increase the number of potential memory channels available as well as provide a large reduction in memory access latency [67].

2.1.2 Going beyond wires

Wires are not the only way to communicate on chip. Recently the possibilities of optical interconnects have been investigated, as well as more exotic wireless communication techniques. Whilst these techniques may not alleviate communication issues to the extent that a single big core can continue to be practical they do provide lower power, lower latency interconnection networks. They can also be used for off chip connections, helping reduce power required for driving external connections as well as decreasing the number of package pins required, allowing greater off chip bandwidth.

New developments in photonics make it possible to produce photonic components in CMOS in a practical way that enables both on and off chip optical communication [36]. In particular on-chip photonics could be used to construct an interconnection network potentially built out of a hybrid of electric and optical connections, 3D integration technology would allow a separate photonic plane to be constructed above the logic plane which performs computation. One such potential design proposed by Shacham et al. [9] when compared to a more conventional electrical only design could reduce network power consumption from around 100W to 4-5W when a large bandwidth is required.

Developments in printed circuit board design [59] allow the inclusion of optical channels along with electric wiring opening up the possibility of optical chip to chip communication. One possibility for this technology would be to use an optical interconnect for memory to creating photonically interconnected DRAM (PIDRAM) as proposed by Beamer et al. [12]. DRAM would be redesigned to use optical channels to connect to the memory controller with full photonic integration inside the DRAM chip (as opposed to using an optical electrical conversion at the memory end and continuing to use conventional DRAM). Such an approach may provide memory with a lower power utilisation as well as allow a bandwidth density two order of magnitude greater that what would be possible with conventional electrical connections. Others [55] have explored the use of optical connections to produce a so-called macrochip where multiple chips are placed within the same package connected via silicon-photonics. The allows the production of a multi-chip system which approaches the performance of a single large chip.

Wireless technology could also be employed to build interconnection networks. Hu et al. proposed [101] a wireless network-on-chip (WNoC) architecture that uses conventional wired networks-on-chip in combination with wireless links. Individual wired subnets can communicate via long-distance wireless links to bypass a multi-hop wired path. This style of design is particular suited to larger networks giving power and latency improvements over a comparable wired only network.

2.2 Interconnect architecture

The shared bus has long been used as a basic interconnection mechanism. Both for chip to chip communication and inter-core communication on one chip. The major advantage of the shared bus is its simplicity, everything that wishes to communicate on the bus simply shares wiring. The protocol controlling bus communication can be very simple as can the arbitration. As only one thing may drive the bus at once a strong ordering on traffic is naturally maintained and as the bus broadcasts information to everything on the bus anything connected to it has global knowledge of bus traffic. However as the number of potential bus masters grows the contention increases, so the bus either needs to handle significantly greater traffic volumes (multiples of the total that could be generated by a single master) or bus masters will be forced to wait. This makes a single shared bus a poor choice for increasingly multi-core systems. Not to mention the engineering difficulties in building a single shared bus that runs over an entire chip that maintains good bandwidth without using too much power.

A natural consequence of the inability to communicate across a chip within a single cycle and the need to best utilise the available wiring to maximise communication efficiency when coupled with multi-core design is the introduction of the on-chip interconnection network [21] or networks-on-chip (NoC). Whilst systems using many processing nodes that require an interconnection network have existed for many years [105], the development of NoCs is relatively new though their design draws upon the design of interchip interconnection networks.

The general structure of a NoC is a collection of routers. Each router is connected to a number of other routers. Something wishing to send a message gives it to a router which forwards it to another router and so on until the message reaches its destination. Messages are split into units known as flits. Sometimes multiple networks may be desired, this can be accomplished by simply replicating multiple physical networks or via virtual channels. A virtual channel is construced by replicating the buffers in a network but not the physical connection. A flit leaving one buffer will be tagged with the next buffer it should be placed in to. By constraining certain messages to only use certain virtual channels a virtual network can be constructed.

For further information please see 'Principles and Practices of Interconnection Networks' by Dally and Towles [22].

2.3 Architectural communication

Architectural communication could be split into two categories, communication triggered explicitly by running code and communication triggered by the architecture to support the running code but that hasn't been explicitly requested by it. An example of the first kind of communication would be in an architecture that has programmatically visible access to an interconnection network. An example of the second kind would be a cache coherency protocol, discussed below.

2.3.1 Cache coherency

In a multi-core system we may wish to have multiple caches and each cache may contain copies of the same memory regions. A problem occurs when cache A and cache B are both caching some memory address x. If the core attached to cache A writes to x and then a core attached to cache B reads from x at some later time we need to ensure the write at A becomes visible to B. This is the coherency problem, we need to ensure writes propagate, when a value is changed it must eventually become visible to others, we also need to ensure writes serialise, changes in the value of the location must be seen in the same order by everyone. Memory coherency is also tied to memory consistency. Consistency concerns the order of operations when dealing with multiple memory locations. Whilst updates to the same location may appear in the same order to all processors, updates to different locations may appear in different orders to different processors. Processor A may see location X change value followed by location Y, but processor B may see location Y change first followed by location X. Coherency and consistency are intimately linked and are affected by the design of a single processor core and the design of the coherency protocol used to ensure cache coherence which in turn is related to the properties of the interconnect used.

Cache coherency protocols can be divided into two categories:

Bus Based Protocols An early example was described by Goodman [34]. Each cache participating in the coherency scheme shares a bus, this bus is used to broadcast coherency messages as well as fetching from main memory (or the next level in the cache hierarchy). Caches snoop on this bus, listening to what the other caches are requesting from memory and potentially intervening (e.g., if cache A has a modified copy of location x and cache B requests a read from x from main memory, cache A could snoop this interaction, interrupt the read and supply B with the latest version of x). A crucial part of these schemes is each cache line has bits giving the sharing state of the line (e.g., a particular protocol may have three states line invalid, line valid but not dirty, line valid and dirty and present in no other caches). These schemes have the advantage of simplicity, however do not scale well with increasing participants due to their broadcast nature.

Directory Based Protocols An early example was described by Censier et al. [68]. Like a snoopy protocol each line has an associated sharing state, but unlike a snoopy protocol this state is held in a central location, the directory. Caches wishing to access a line can consult the directory to determine the state of the line and potential sharers of the line avoiding the need for a broadcast. These schemes may be more complex to implement than a snoop shared bus scheme (they require a proper interconnection network rather than a shared bus) but can scale with increasing participants [2]. However to maintain good scaling the directory needs to be distributed to avoid a single directory becoming a bottleneck for the whole system.

Sharing states

Both directory based and bus based snoopy protocols need to keep track of the sharing state of a particular cache line (or potentially some larger region of memory as explored in [14]). This sharing state drives the entire coherence protocol. The most basic example would be a valid, invalid system. Here a line being marked as valid means that it is present in strictly one cache, and an invalid line is in no cache. In a snoopy system when a cache has a line in a valid state any read or write requests to that line will result in a hit, if a cache does not have that line it will request it from main memory. All caches snoop the memory access and if a request is for a line in a valid state that cache will interrupt the

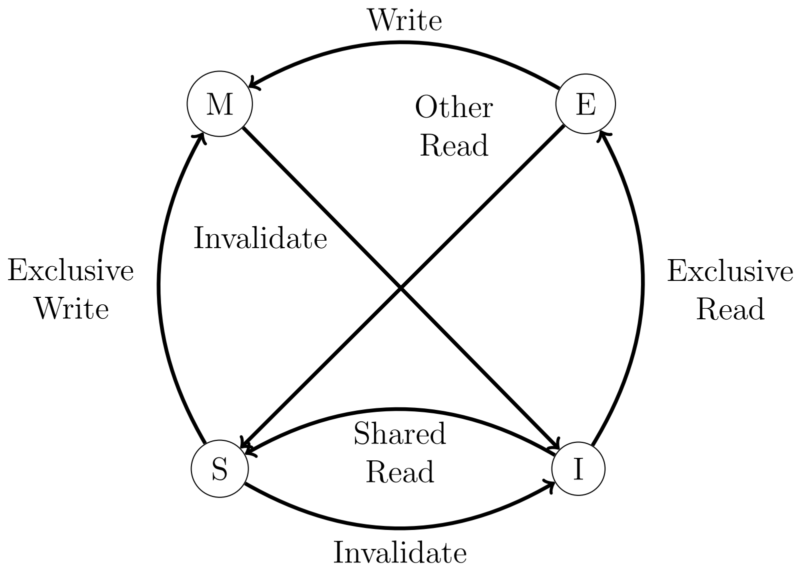


Figure 2.1: A MESI cache coherency protocol state transition diagram

memory request and either supply the line directly to the requesting cache (switching its version of the line to invalid, the requesting cache will store the line in a valid state) or first write the line back to memory (so the line will not be held in any cache) and then the requesting cache will retry and get the line from main memory storing it in a valid state.

In a directory based system with states (valid, invalid) a cache that doesn't hold a line that it needs to service a read or write will first check the directory, this will state whether the line is valid, meaning it is in another cache or invalid, meaning it is in no cache and can be fetched directly from memory. In the case of the line being in another cache the protocol proceeds as the snoopy protocol above (either the cache holding that line writes back or directly transfers to the request cache). The key difference being the directory removed the need for broadcast. It is important to note that state is still required within the cache, it needs to know whether a line is in the valid or invalid state so it knows whether or not it holds the line.

Whilst very simple, the valid, invalid scheme is poor. An obvious deficiency is the inability for a cache line to reside in multiple locations, in the case of rarely written but frequently read data much unnecessary bus traffic is generated. The addition of a third state to give a modified, shared, invalid (MSI) protocol can solve this. An invalid line exists does not exist in a cache (but could be in another), a shared line exists in one or more caches but is identical to the line in main memory and a modified line exists in only one cache and is an updated version of the line in main memory (it is dirty).

A more advanced scheme, which more closely resembles the protocols used in contemporary architecture is the MESI protocol [83]. This protocol utilises four sharing states, modified, exclusive, shared and invalid. The meaning of the states is the same as the MSI protocol with the addition of the exclusive state. A line in the exclusive state is unmodified (identical to the line in main memory) but is in only one cache. A key feature of this protocol is it allows a line to move from exclusive to modified (due to a write hit on that line) without the need for any communication. In MSI a write hit to a line in the shared state requires the cache to first invalidate that line (either by broadcasting an invalidate message or using the directory to determine which caches to send it to) to complete the write even if nothing else is sharing it, MESI avoids this potentially wasteful communication. A simplified state transition diagram for MESI can be seen in Figure 2.1. In the figure an exclusive read is one where the location being read is in no other cache, a shared read is one where the location being read is in another cache in the shared state, an other read is where another cache is reading something that exists in this cache, an exclusive write is one that sends an invalidate message to any other caches that hold that location.

Further developments

Many variations and extensions to the above have been proposed, further states could be added (e.g., an Owned state to indicate modified data, that is present in multiple caches to give the MOESI protocol as proposed by Sweazey et al. [95]), optimisations can be done on directory structure (e.g., using coarse notions of what caches are sharing a line combined with sparse directory structures to minimise storage and network traffic needed to implement a directory based scheme as proposed by Gupta et al. [7]), specific features could be added to the interconnection network to support coherency (e.g., employing multicast support so sharers of a line can multicast directly to each other without the need to consult a directory as proposed by Jerger et al. [30]), speculative communication could be employed to improve miss times (e.g., communicating with neighbouring caches to find a line before going to the directory as proposed by Barrow-Williams et al. [11]) as well as novel entirely new coherence methods (e.g., token coherence where a number of tokens are associated with each cache line, a cache may read from a line if it holds at least one token, but may only write to the line if it holds all tokens as proposed by Martin et al. [69]).

A unifying theme amongst all cache coherency protocols is the need to preserve the illusion of a global shared memory amongst all cores to the programmer. A protocol observes the loads and stores executed by a core and attempts to best optimise the communication amongst caches so it can be kept to a minimum whilst also offering good hit rates and lowering miss penalties. Crucially the programmer has no explicit control over this, they may only arrange and process their data in a way that best suits the protocol being used in the hope that the communication that occurs on the underlying interconnect will not be wasteful.

2.3.2 Message passing

Most modern multi-core architectures utilise a shared memory programming model implemented via cache coherence. If a message passing facility is desired it is generally implemented via software, though several direct hardware implementations exist which give programmatic control over communication. An early example is the Transputer [103]. Each core had a series of point to point links available accessible via special instructions.

Communication was entirely synchronous, messages were fixed in length and a core sending a message had to wait for an acknowledgement from the receiving core before sending the next message, a direct hardware implementation of the Occam [72] programming model.

The RAW architecture from MIT [96] is a more recent example of an architecture with hardware support for message passing. It utilises a far more advanced design than the transputer connecting multiple cores on a single chip with four NoCs. Two of the four NoCs had purely static routes, defined at compile time, the other two had dynamic routing. The networks are exposed to software via special instructions as well as direct register mapping. Reading and writing to certain registers directly reads and writes to the incoming and outgoing FIFO buffers of the network routers. Two dynamic networks were required to protect against deadlock. One network is for use by privileged code only and the other for general usage. Anything using the privileged network ensures it follows certain rules to avoid creating a protocol deadlock in the network. The general network may deadlock due to improper usage, this can be detected and then recovered from using the privileged network.

Sanchez et al. recently proposed Asynchronous Direct Messages (ADM) for use in implementing fine-grained scheduling [87]. ADM is used as an extension to an architecture already using cache coherency with an interconnection network used to run the protocol. A virtual network is added to the existing network for the purposes of passing messages. Messages can be received asynchronously or synchronously with all functionality handled by new instructions. Asynchronous reception is implemented via an existing exception handling mechanism. ADM buffers messages in hardware and guarantees both message delivery and order. A full hardware buffer is handled by trapping to software and having it store the second half of the full buffer elsewhere and returning it to the buffer when the first half has been received.

HAQu proposed by Lee et al. [88] is a hardware accelerated mechanism that can be used to implement message queues on top of a cache coherent architecture. Given software queues with a certain layout in memory HAQu provides a number of new instructions that operating upon the queues implementing a message queue algorithm based upon one presented by Lee et al. [60]. This approach has the advantage than no extra interconnection network (virtual or otherwise) is required on top of the existing one required for the cache coherency protocol (with the attendant extra hardware resources required and potential deadlock issues this creates) and achieves good speed-ups compared against a purely software implementation (6.5-7x) however unlike direct hardware scheme the actual communication over the interconnection network is still controlled by a cache coherency protocol and cannot be directly controlled by software.

2.3.3 Consistency

An issue with any multi-core system using a shared view of memory between cores is consistency. Given all of the loads and stores executed by all cores there should be some way to arrange them (in a program order preserving manner so loads and stores from thread A will be executed in the order specified by thread A, but stores and loads from other threads may interleave them) such that the result of executing that arrangement on

```
void threadA() {
        threadAIn = 1;
        if(threadBIn == 0) {
                //Critical section
        }
}

void threadB() {
        threadBIn = 1;
        if(threadAIn == 0) {
                //Critical section
        }
}
```

Listing 2.1: Dekker's algorithm for mutual exclusion, initially threadAIn == threadBIn == 0, note this is purely used as an example to illustrate consistency, details of how a thread exits a critical section or waits the other thread to exit need not concern us

a single memory in sequential order will give the same result (in terms of final memory contents and data returned by loads) as was actually observed, this is known as sequential consistency [1] and is a simple intuitive memory model for a programmer to work with. Unfortunately sequential consistency is not always possible to maintain, because it prevents various optimisations (such as store buffers discussed below). Consider the example of Dekker's algorithm for mutual exclusion in critical sections taken from [1] shown in listing 2.1. With a sequentially consistent programming model when thread A reads threadBIn if it finds it to be 0 then that means that thread B hasn't executed its store to threadBIn yet so its safe to enter the critical section because threadAIn has been stored to and when thread B executes its load of threadAIn it will be prevented from entering the critical section. A relaxation of sequential consistency could break this, for example if thread A's store to threadAIn was in some way delayed from being seen by thread B then it would be possible for thread A to load 0 for threadBIn and continue to the critical section and for threadB to do the same and continue to the critical section (as A's store to threadAIn may not have reached thread B's view of memory).

Fundamentally relaxed consistency models occur because all cores do not share a single block of memory giving a single view of memory (they have separate caches, and there may be multiple banks of memory) and the communication that a particular core has executed a particular store on a particular block is not instantaneous. As a concrete example a common architectural optimisation is the addition of store buffers, when a core executes a store it places it in a buffer pending completion so execution can continue without needing to wait for the store to finish. Stores are still executed in program order but the result of a store will not be immediately visible to other cores (later loads on the same core from the same location will peek into the store buffer before accessing cache or memory). A two core system with a shared bus MESI cache coherency protocol with store buffers at each core may not execute Dekker's algorithm correctly. For example if both cores had threadAIn and threadBIn present in their cache in the shared state then the loads of threadAIn and threadBIn could complete immediately whilst the stores to threadAIn and threadBIn may sit in store buffers for some amount of time. This would allow both threadA and threadB to enter the critical section.

One way to deal with this issue is the introduction of memory barriers or fences [84].

A memory fence will cause a core to wait until memory operations (or some subset of them) proceeding it have actually completed. There are many different types of fences, the precise types provided depend upon the consistency model used. As an example in the hypothetical two core system with shared bus MESI cache coherency discussed above a fence could be introduced that waits until all stores currently in the store queue have had their write invalidate messages acknowledged (which means the associated cache line is marked invalid in the cache that didn't broadcast the message). Dekker's algorithm can be made to work again by inserting such a fence after the store. The ensures that when the load occurs if the thread receives a 0 from the load and so continues to the critical section the other core either hasn't done its store or hasn't complete its memory fence yet.

Another approach is to enforce sequential consistency via novel methods that do not impose large performance penalties (many consistency issues occur because of architectural optimisations, such as store queues, simply removing these could restore sequential consistency but the cost to lost performance is generally considered to be too high). Ceze et al. propose Bulk SC [16] which divides loads and stores into chunks, each chunk appears to execute atomically and in isolation, sequential consistency is then enforced between chunks to give overall sequential consistency. It accomplishes this by associating read and write signatures with each chunk which can be compared between chunks to determine if they perform loads or stores to the same locations. If they don't interfere with one another it doesn't matter how they are executed with respect to one another. Sequential consistency only needs to be enforced amongst chunks that do interfere. BulkSC can deliver similar performance to relaxed consistency models without too much extra overhead (5-13% extra network traffic on average).

2.4 Software communication

The two major models used in software for multi-core and multi-processor systems are the shared memory model and the message passing model. In a shared memory system all threads can have access to the same address space, in a strict message model system threads only have access to some local address space and must communicate with other threads via messages. The shared memory model may appear easier to program initially but the lack of isolation between threads causes many issues. The message passing model does not suffer so badly due to the stricter isolation that is enforced by having separate address spaces.

Distributed shared memory (DSM) systems implement a single virtual address space on a system consisting of multiple separate physical address spaces belonging to different processors [79], they perform this at different levels of granularity. Whilst running a software based DSM system on a page-level of granularity could be practical a software based DSM system running at smaller levels of granularity (e.g., a cache line, effectively running a cache coherency protocol in software) is not.

The converse, building a message passing system in software on top of a shared memory system (utilising cache coherency protocols in hardware), is commonly done. Much research has been done on how to build efficient message queues on top of a cache coherent system [61, 37]. Some programming environments enforce strict thread isolation, allowing only message passing as the sole communication mechanism between threads (such as Erlang [8]). Message passing is also a useful primitive in its own right and is often used with other methods in shared memory programming models.

2.4.1 Locks, barriers and notification

In a pure message passing system synchronization between threads can be achieved implicitly. If a thread should not continue past a point before another thread has accomplished some task it can wait for a message from that thread. Atomicity of access to data structures is of no concern because of the strict memory isolation between threads. In a shared memory programming model primitives are required to explicitly enforce synchronization.

Locks are widely used to regulate access to critical sections in shared memory programming. The use of locks introduces a range of issues that the programmer needs to consider when using them such as the possibility of deadlock. Another consideration with lock based programming is granularity, how many locks should be used? Coarser granularity may be simpler to reason about but a finer granularity of locking promotes more concurrency however it may exacerbate deadlock issues and if locks are too fine grained much time will be wasted acquiring and releasing locks. With increasingly multi-core systems software must utilise more concurrency in order to get the best performance out of an architecture, efficient fine-grained locking can enable this.

Alternatively locks could be abandoned altogether. Lock-free programming utilizes a small set of primitive atomic memory operations (e.g., compare and swap, which was shown by Herlihy to be a universal primitive [40] in the sense that it can be used to implement any lock-free structure) provided by hardware (these primitives could be built upon to implement more flexible and complex atomic operations such as a multi-word compare and swap as proposed by Fraser et al. [33]) to implement data structures that can be accessed concurrently without needing locks. This allows fine-grained concurrency as well as removing deadlock issues, however correct lock-free data structures require careful construction.

Barriers are another key synchronization primitive which again presents a granularity issue. Say an algorithm proceeds in several steps, a barrier may be needed so each thread starts each step together after all threads have finished the previous one, but there may be certain subsets of threads that could continue past the barrier once every thread in that subset has finished so a more fine-grained approach to the barriers would allow a greater degree of concurrency. Much like using more fine-grained locks, more fine-grained barriers will increase program complexity and more time may be wasted executing the barrier. An ideal barrier system would be one that allows a thread to continue as soon as precisely the data it needs is ready, rather than having to wait for all threads, building a data-parallel model of execution.

Fundamentally any synchronization technique requires some form of notification, an ability to signal to a thread than an event has occurred. For locks this event will be the locking being released, so something else may acquire it, for the barrier the event is every expected thread reaching the barrier. In message passing system a thread needs to know when a message has arrived so it can continue if blocked on a receive operation. Even in lock free programming a notification mechanism could be useful, a construct often used in a lock free programming is to loop attempting an atomic operation such as a compare and swap. If the operation fails then it is tried again until is succeeds, because some thread must succeed progress is guaranteed. However in a high-contention situation several threads may attempt multiple retries before succeeding, wasting CPU time and potentially causing needless communication. A notification mechanism that could notify a thread when it is clear to proceed could be very useful.

2.4.2 Partitioned global address space

The partitioned global address space (PGAS) model is one where each thread is associated with a particular part of address space. The associated memory may be private so only the associated thread may access it, a separate global memory is provided for inter-thread communication. The unified parallel C language from Berkeley [97] implements this memory model. This approach has the advantage that the programmer has explicit control over data placement as well as greater knowledge of when communication occurs (a write to a private space doesn't require communication, a write to a shared global space does, though if the shared space is implemented with a standard cache coherent system the timing and precise nature of the communication is still obscured). A PGAS model where the partitioned space isn't private to a particular thread (removing the need for the separate global shared space) is also possible. Here accessing a thread's own memory partition would be cheaper (in terms of communication incurred) to access than another thread's memory partition, thus by choosing where to place data and when to access it the programmer could exercise control over communication costs.

Related to the PGAS model is the remote store programming (RSP) model proposed by Hoffman et al. [39]. In RSP each process or thread is created with a purely private address space to start with. Part of this address space can be declared writable by remote processes or threads, communication is then accomplished by writing to remote address spaces. By arranging private address space so it is cheap (in terms of communication) to access from the thread it is associated with a programmer is able to optimise for communication. Hoffman et al. found that RSP allows the creation of programs that scale well with increasing core count as well as achieving higher performance than a standard shared memory implementation of some applications.

2.5 Existing architectures

Many multi-core designs have been produced and are in active use. Several examples are discussed below in the context of the section above. Physical interconnect is not discussed in the examples below. Whilst many interesting innovations in silicon technology have been required to produce the example below they all use electrical interconnect over wires as the more exotic methods haven't progressed beyond the research stage and discussion of the specific developments required to make the wires work are beyond the scope of this dissertation.

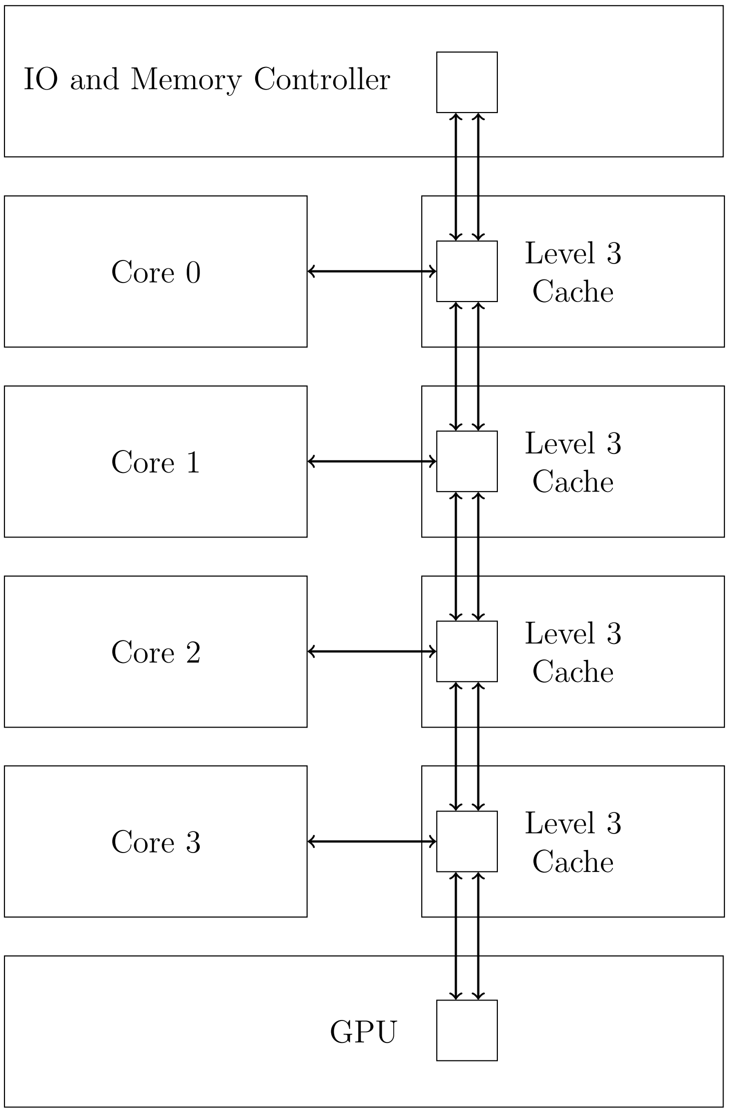


Figure 2.2: Sandy Bridge Architecture

2.5.1 Sandy Bridge

Sandy Bridge is a multi-core microarchitecture created by Intel targeted at a 32nm process. It is used in consumer, business and server class machines. A block diagram of the general layout can be seen in Figure 2.2. It can be configured with different numbers of cores (up to 8 in currently released versions), the figure shows an example with 4 cores. Each core is a out of order superscalar SMT core (so is capable of running more than one thread at once) executing the x86 instruction set including extensions for SIMD computation of floating point vectors. [106]

Interconnect

Interconnect on Sandy Bridge is provided by a bi-directional ring, this connects the cores, L3 caches, IO and (on-chip) GPU. Off-Chip interconnect is provided by Quick-Path Interconnect (QPI) [46] which consists of high speed point to point links that allow separate chips to communicate. It uses virtual channels to support 3 separate virtual networks for different message classes, credit based flow control is used chip to chip. Routing is handled by static routing tables defined in firmware allowing a variety of topologies.

Architectural communication

Sandy Bridge provides a shared memory model to all cores using cache coherency. Each core has a private L1 and L2 cache, the L3 cache is banked and is shared amongst all cores in a single chip. A MESIF (MESI with an additional F, forward, state) directory based protocol is used. The forward state is similar to the shared state, but a cache with a line in the forward state must supply that line to another cache when it requests that line [45] (this is an optimisation to prevent all caches with that line providing it wasting bandwidth). The QPI protocol is used to carry coherence traffic around the ring on-chip, off chip the QPI interconnect is used to carry coherence traffic. Sequential consistency is not maintained so a number of memory fences are available. So-called locked instructions are also available, these can be used as base atomic primitives (e.g., compare and swap) and they are defined to execute in a total order that is visible to all cores, effectively giving sequential consistency amongst all locked instructions. No formal official definition of the x86 consistency model exists but Owens et al. [82] have created one based upon available documentation. Briefly Sandy Bridge has the same ordering characteristics as an abstract machine where each core reads and writes to a single block of memory but writes may be buffered (so are delayed from being applied to the main memory). A core executing a read will obtain the latest data for the address being read from its own write buffer, but not from any other cores write buffer (that is writes from another core will only become visible after a certain time).

Software

In general the x86 instruction set architecture is widely used for a large number of applications, Sandy Bridge is no exception. A conventional shared memory with locks programming model is often utilised in Sandy Bridge systems but a variety of other models, as discussed above, may also be seen.

2.5.2 Tile Architecture

The Tile Architecture is a product of Tilera. Tilera offers a variety of multi-core products with this architecture, one of which is TILE64 which contains 64-cores on chip, the initial implementation was done on a 90nm process. A block diagram of the general layout can be seen in Figure 2.3. Each core implements a VLIW instruction set and may run an operating system independently of the other cores [102]. The architecture is flexible so products with large variations in core number can be offered without any architectural revisions. The Tile architecture is used in a variety of applications requiring large amounts of concurrency, such as server, multimedia and networking applications and was inspired by the MIT RAW architecture [96].

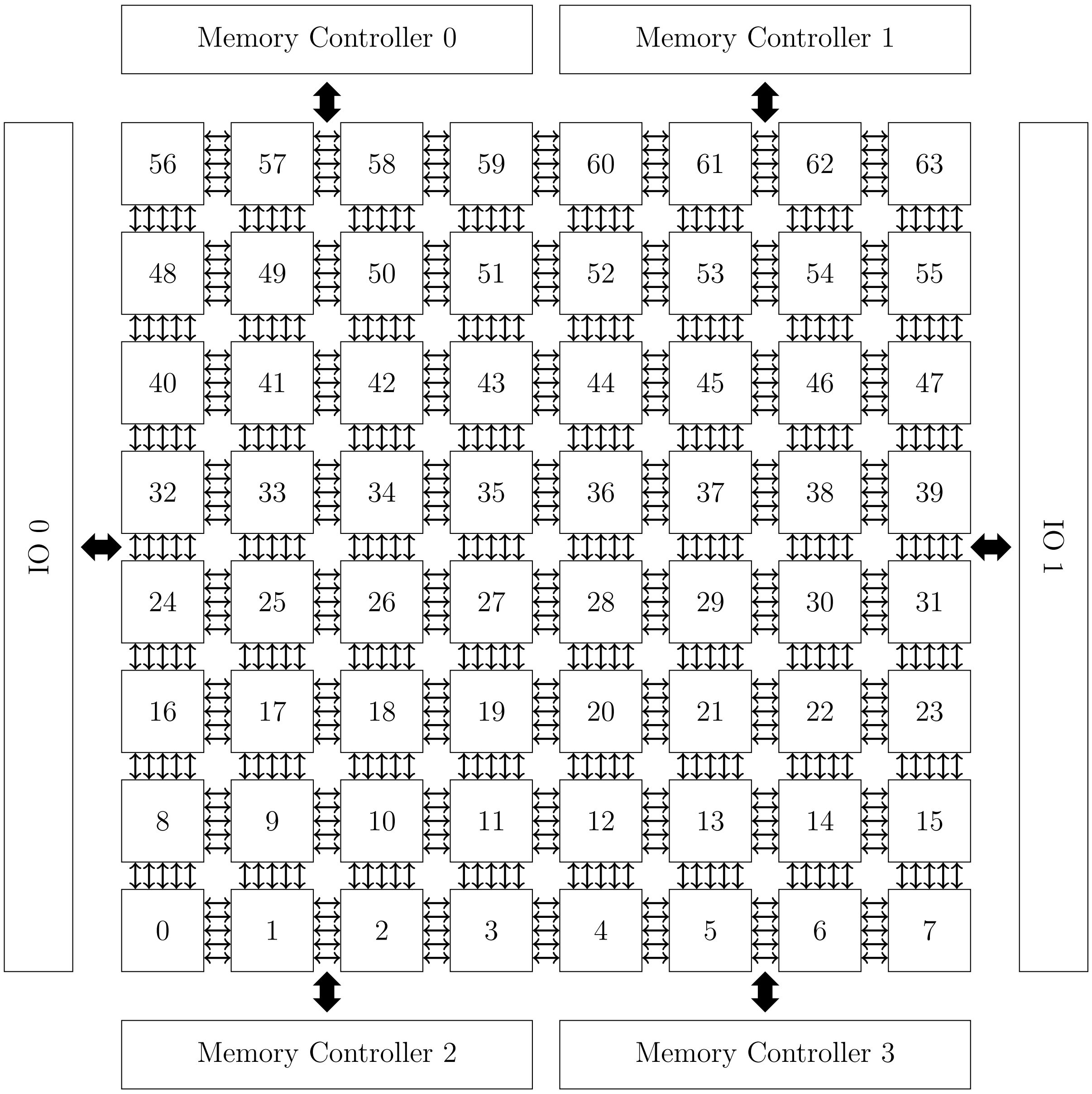


Figure 2.3: TILE64 Architecture

Interconnect

Cores in the Tile Architecture are connected with a 2D-mesh topology, there are five separate networks, one of which is statically routed. The dynamic networks all use dimension ordered routing. The four dynamic networks have separate uses. One is the user network which is directly accessible in software, I/O access is done over another network, allowing any core to talk to any of the attached I/O devices. A fully cache coherent shared memory is also provided, this uses two networks, one, the memory network, is for cache to memory controller communications which works with the other, the tile network, which carries coherence traffic. Cache to cache transfers are possible and require the use of both the memory and tile networks for responses and requests respectively to avoid protocol deadlock. Interestingly virtual channels are not employed, each network is a physically separate network.

Flow control is a credit based scheme. The memory network implements further end to end flow control to avoid deadlock, a node is allocated space in a buffer at a memory controller and must never use more than its allocated space. The user and IO networks have mechanisms to drain and refill the network so in the case of a deadlock it can be recovered from.

Architectural communication

Coherency in the Tile Architecture is provided by a directory based cache coherency protocol. Each core has an L1 and L2 cache, the coherency protocol runs between the L2 caches. The TilePro variation of the architecture allows software to specifically 'home' a page of memory in a particular cache. If a cache line is homed like this then any write accesses or any read misses are always sent to the cache where the line is homed. This allows software to localise a particular part of memory to a particular core and cache, cores close to the home location will incur less communication costs to use that location than cores further from the home location, giving the programmer some explicit control over communication costs. Consistency is similar to the Sandy Bridge above. Writes will be visible immediately to the core that performed them but may not be immediately visible to all cores. When a write does become visible to all cores it becomes visible to all cores simultaneously, a memory fence instruction is provided as is an atomic test and set operation. Test and set operations become globally visible to all cores at the same point, so there is a single total ordering of them visible to all cores. [98]

The architecture allows portions of a core's L2 cache to be used as a scratchpad memory, a software programmable DMA engine can be used to move memory between theses scratch pads as well as to and from main memory. This gives direct programmatic control over incurred communication [98].

Another way for the programmer to have direct programmatic control over incurred communication are the user and static networks which allow software to communicate directly over the interconnection network between cores. This is combined with a fast interrupt mechanism that can be used by non-privileged code which is used to notify threads of data arrival on the dynamic user network. The networks are register mapped with a direct connection into the ALU allowing rapid communication between cores [98].

Software

Each core in the Tile Architecture is capable of running Linux separately, or several cores can be used to run a single instance of SMP Linux. So a wide variety of programming models can be used (including a conventional shared memory with locks model). However to best utilise the Tile Architecture software needs to be written with the interconnection network in mind. The static network can be utilised to exploit data parallelism and to build software pipelines, this could be done explicitly by a programmer or a compiler could schedule the instructions of a sequential program across several cores [63]. For utilising the dynamic network Tilera provide a library iLib that provides multiple channel abstractions. The dynamic user network can be exposed as raw channels (reading and writing directly to the network's buffers), buffered channels (adding software buffering and control on top of the raw network channels) or as a generic message passing interface.

The remote store programming (RSP) model developed by Hoffman et al. [39] has been implemented on TILEPro64, where the ability to home regions of memory at a particular cache is utilised. The core running a particular thread has that thread's private memory homed at that core's cache, so when part of that memory is made available for writing the cost of another thread writing to it is directly proportional to the distance between the cores running the threads in the network. Unified Parallel C has also been implemented on the Tile Architecture again exploiting the locality exposed in the language [89].

2.5.3 Single-chip Cloud Computer

The Single-chip Cloud Computer (SCC) is an experimental prototype chip produced by Intel Research. Its purpose is purely to enable research into multi-core design so it doesn't target any particular application area. Each chip contains 48 cores and is implemented on a 45nm process. A block diagram of the general layout can be seen in Figure 2.4. Each core is in-order superscalar, based on an old Pentium design [44].

Interconnect

Cores in the SCC are connected in a 2D-Mesh. Two cores are coupled together to share a single router forming a tile. Dimension ordered routing is used with credit based flow control. The network provides eight virtual channels two of which are reserved for request and response message classes for deadlock avoidance. Each tile contains a message passing buffer (MPB), an SRAM memory to aid in the implementation of coherency and message passing [44].

Architectural communication

The SCC does not implement cache coherency in hardware. Each core has private L1 and L2 caches, a miss in both of these will be serviced by one of the four DDR channels with no interaction between the caches of other cores. A look up table in each core divides the memory into 256 partitions, each partition can be pointed at physical off-chip memory (DDR) or at a tile's MPB (there are also some memory spaces used for configuration and communicating with the external management system). By manipulating the look up tables in each core physical memory can be divided in shared and private areas, a bit in the page tables can mark memory as non-cacheable, this could be used for a shared memory area so the lack of cache coherence isn't problematic though it has obvious performance issues. Another bit in the page table is MPBT. When this is set memory belonging to that page can only be cached in L1, and there is a special instruction to invalidate any lines belonging to an MPBT tagged line in the L1 cache. Combined with the MPB memory this can be used to implement message passing. An MPB will get mapped via the lookup table to a certain set of pages, these will be marked as MPBT. To

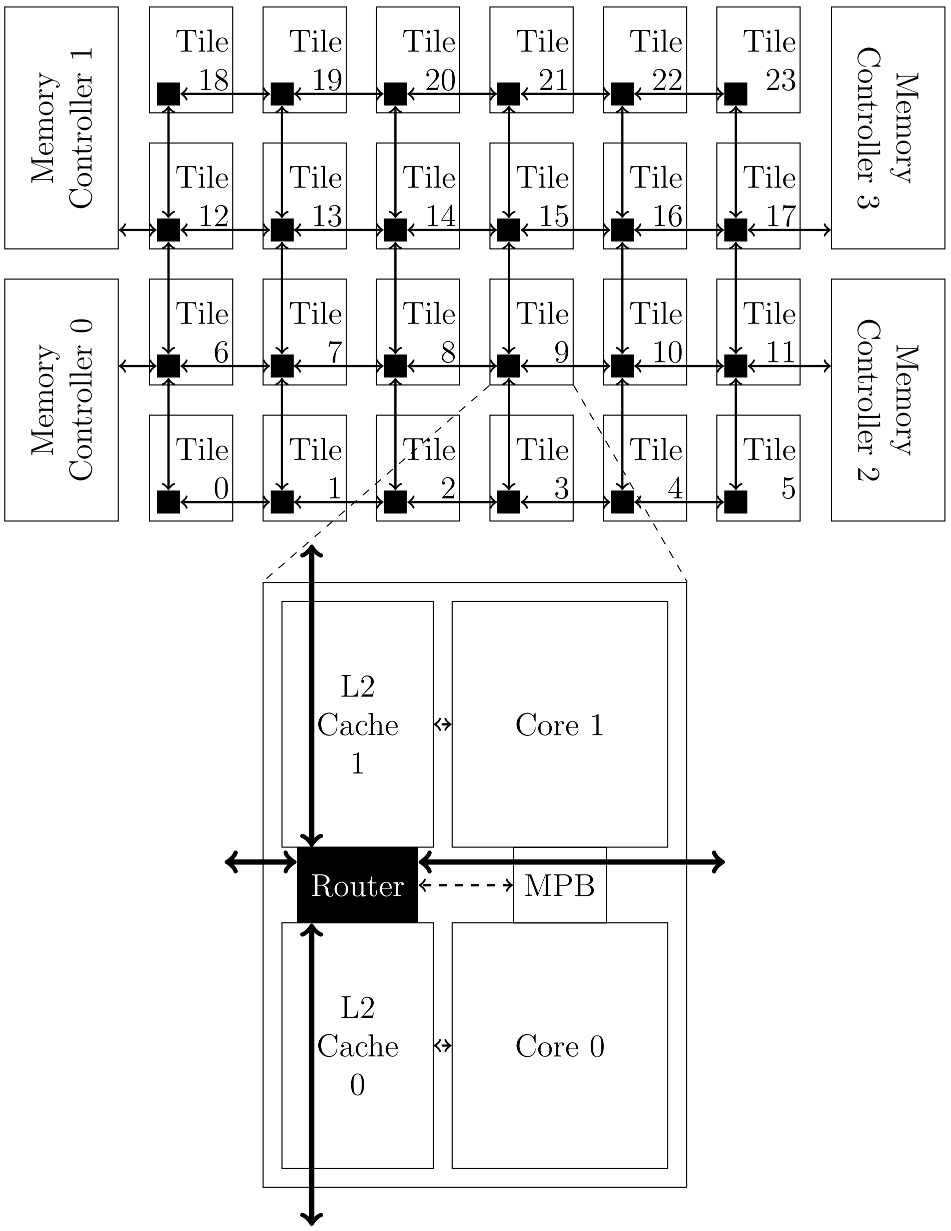


Figure 2.4: Single-chip Cloud Computer Architecture

send a message first invalidate all MPBT marked lines in the L1 cache. Writes to those lines will get passed through the L1 direct to the destination MPB (a write combining buffer is used to achieve good performance). The destination core can read its tile's MPB through the L1 cache and then invalidate the MPBT marked memory in its L1 to ensure it doesn't read stale data the next time [50].

Software

Whilst each core in the SCC runs the x86 instruction set and so is able to run the wide variety of software available for x86 the lack of cache coherency means that software not designed for the SCC can only run on a single core unmodified. An SCC system may

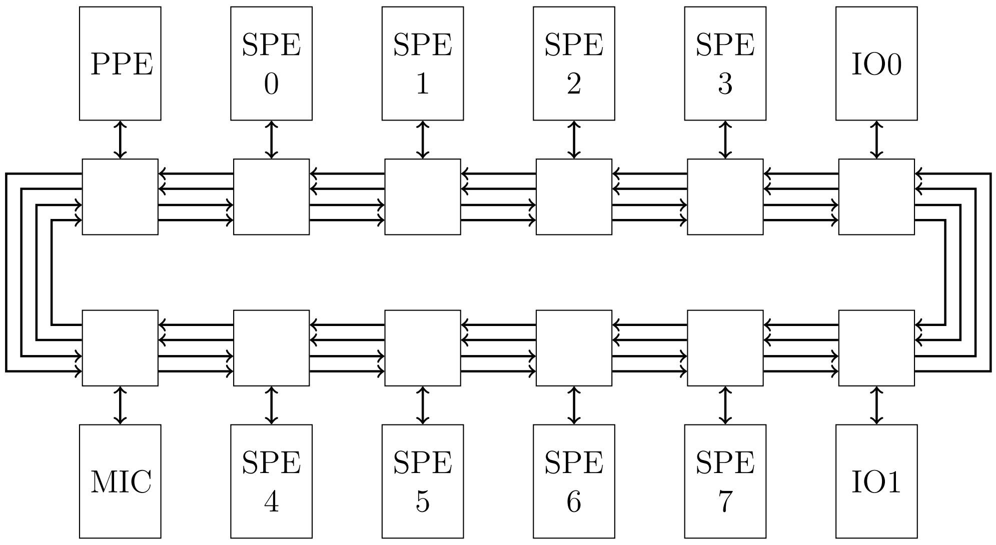


Figure 2.5: Cell Broadband Engine Architecture

start separate Linux instances on each core which can then communicate via message passing, Intel provide a library RCCE that effectively provides a light wrapper over the MPB functionality allowing programmers to put and get data to and from any MPB in the system along with some primitive synchronization methods so cores can wait for other cores to complete an MPB operation before they begin their own [70]. Clauss et al. [19] implemented an improved version of RCCE, iRCCE, which divides MPBs into two chunks, allowing one core to write into an MPB whilst another cores allowing an overlapping of execution for those cores when they are using the MPB to communicate (with RCCE a core would have to wait for the other core to fill or drain the entire MPB before continuing). iRCCE was compared to an MPI implementation also written by Clauss et al. and using more conventional shared memory techniques. Purely using shared memory offer predictably poor performance as shared memory must be uncached, iRCCE offered the best performance. Still further cache control (specifically the ability to flush an L2 cache) was desired, the explicit communication that is offered by the MPB mechanism was not sufficient to get the best performance out of the processor.

2.5.4 Cell Broadband Engine

The Cell Broadband Engine is a heterogeneous multi-core architecture developed by IBM. It consists of 9 separate cores, one is an in-order superscalar core compliant with IBM's Power architecture capable of running 2 threads at once known as the power processing element (PPE). The other cores execute a pure SIMD instruction set running on wide (128-bit) floating point vector registers, these cores are known as synergistic processing elements (SPE). The major target area of the Cell is games and multimedia with an emphasis on real-time responsiveness [53]. A block diagram of the general layout can be seen in Figure 2.5. Along with the PPE and SPEs the Cell contains 1 memory interface controller (MIC) and two IO controllers.

Interconnect

Four unidirectional rings provide the interconnect in the cell (two go clockwise the others counter clockwise). The rings connect together all 9 cores, the memory controller, and two IO interfaces. One of the IO interfaces can be used to connect to another Cell processor connecting a total of 2 PPEs and 16 SPEs together into the same network [54].

Architectural communication

The PPE is a standard Power Architecture processor with a L1 and L2 cache. The SPEs do not have cache or direct access to main memory. Each SPE has a small (256 KB) local memory which it operates on directly. Each SPE also has a memory flow controller (MFC). The MFC contains a DMA unit that can be used to transfer data between the SPE's local memory and main memory and transfer data between the local memories of two SPEs. The MFC allows multiple DMA commands to be queued up and executed whilst the SPE computes in parallel. This allows a streaming style of computation with the MFC streaming data in, the SPE performing computation on it and the results being streamed back out by the MFC again, either to main memory or to another SPE for further processing [54]. DMA operations may be completed out of order so fences are provided to ensure order when this is required. Special DMA operations are provided to perform atomic operations on lines in memory.

In addition to the MFC and local memory each SPE has two signalling channels and a set of mailboxes, an SPE can block and wait on a signalling channel or for a message to arrive at a mailbox. The other ends can be signalled or written by another SPE or the PPE. A write to an outbound mailbox of an SPE can generate an interrupt on the PPE.

Software

The PPE can run Linux, however using the PPE on its own is clearly not the best way to utilise the Cell, effective use of the SPEs is required to get the best from the Cell architecture. Due to the specialised nature of the SPEs running a multi-core capable operating system (such as Linux) over all 9 cores and using a shared memory and locks programming model is impractical (for one some kind of software coherency and caching scheme would be needed). A programmer could write software directly targeted at the Cell, an approach taken by Chow et al. [18], this provides excellent performance (Chow et al. achieved 46.8 GFlops on their FFT implementation running on a Cell running at 3.2GHz compared to an optimized FFT implementation running on a Power architecture machine at 1.65 GHz which only achieved 1.55 GFlops) but this requires a large progamming effort. Ohara et al. [81] proposed a new programming model, MPI microtask, based on MPI. A programmer must partition their program into microtasks that can fit into an SPE's local store. The system then handles scheduling these microtasks and the required memory movement. Eichenberger et al. [29] proposed various compiler techniques that can be used to automatically parallelise a program across SPEs when that program is written with an OpenMP model achieving a 7.1x speedup.

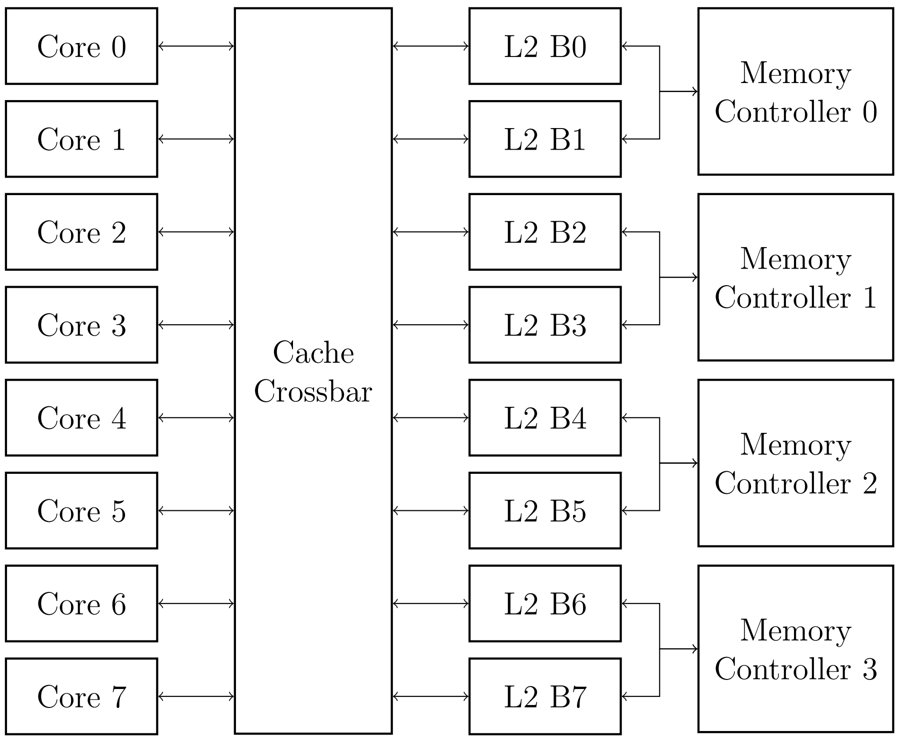


Figure 2.6: Niagara Architecture (UltraSPARC T2 pictured, IO interfaces not shown)

2.5.5 Niagara Architecture

The Niagara Architecture was created by Sun. It is targeted at commercial server class applications that tend to have a low degree of instruction level parallelism (ILP) but a high degree of thread level parallelism (TLP) [56]. Figure 2.6 shows the general layout of the architecture. The actual chip pictured is the UltraSPARC T2 [90] an evolution of the original implementation of Niagara the UltraSPARC T1. Each core is a 64-bit SPARC processor core capable of executing eight hardware threads independently. These are implemented in a fine-grained manner. Every cycle the thread select stage of a core's pipeline chooses one of the eight threads to execute an instruction for. An instruction from the selected thread is fetched and pushed into the beginning of the pipeline. The thread selection policy aims to switch threads every cycle giving a fine-grained interleaving of threads in the pipeline.

Interconnect

The Niagara uses a crossbar for its interconnect. This crossbar connects every processor core to the eight level 2 cache banks (L2 B0 etc in the diagram), but does not allow direct communication between two cache banks or two cores. The crossbar allows up to eight simultaneous accesses of the level 2 cache provided they each use a different bank. Four separate memory controllers provide the level 2 cache banks with access to main memory.

Architectural Communication

The Niagara is a shared memory machine that implements a directory based cache coherency protocol. All of the (private, per core) L1 caches implement a write through policy and an L1 line is either in a valid or invalid state. A particular L1 line will always be backed by the same L2 cache bank so due to the write through policy the L2 cache can maintain a sharing directory of which L1 caches share a particular line. When a core stores to a particular word in memory, this store will be written through the L1 cache to the L2 which allows the L2 to invalidate all of the L1s that share the updated line. A store will not update the L1 cache until the L2 has been updated, before the L2 has been updated only the thread that issued the store can see the new value. This ensures that a store from a thread becomes globally visible to all other threads at the same time regardless of which core they are running on.

Software

Nigara is designed to run the Solaris operating system. The processor appears as 64 discrete processors to an application, the OS abstracts away the details of how these are mapped into hardware threads and cores. Any application designed to run on Solaris will work without any need for modification but only those that utilise multithreading will gain any benefit from the architecture.

2.5.6 Tile based, application dedicated architecture

An emerging category of processor architectures is that of tile based architectures that are intended to run a specific application. For example video decode is highly parallelisable and maps well to a grid of tightly coupled processing elements. Dedicated hardware is often constructed for the purpose but building an ASIC for a particular application is expensive and leads to long development cycles. An emerging alternative are lightweight processors that consist of many simple processing elements networked together. One example is the picoArray architecture [27]. Whilst it is generally programmable it is designed for targeting specific highly parallelizable applications that will be the only thing utilising the chip.

A picoArray contains 430 processors, connected in a grid network consisting of many unidirectional buses. The buses are interconnected via switches which are programmed by software to have a particular switch schedule. Time is divided into slots and for a particular slot particular paths will exists between processors as determined by the switch schedule. A processor can send and receive messages directly via the network, either synchronously, where neither sender or receiver may proceed until the communication is complete or asynchronously, but this carries the risk that a message may be missed as it gets overwritten in an incoming buffer before the receiving processor gets a chance to look at it. Each processor is a 16-bit 3-way VLIW core each with their own small local memory (between 1KB and 32 KB). Special purposes peripherals, such as an SRAM interface or chip-chip interconnections are also present, which are accessed via the same network processors use to communicate.

A related architecture is the XS1 [71]. Again it consists of an array of simple processor cores interconnected by a network. Though unlike the picoArray the network has some routing functionality, a message header can describe a message's intended destination and be switched appropriately. The processor used by the XS1 is also more sophisticated, it has hardware support for executing a number of threads. Fine-grained multithreading, similar to what is found the Niagara architecture is utilised to execute multiple threads. Several instructions in the XS1 cause a thread to stall pending some external event. Due to the fine-grained nature of the thread scheduling this allows extremely rapid wakeup of a stalled thread. This provides excellent responsiveness allowing the XS1 to implement systems in software that traditionally required a hardware approach due to their real-time constraints.

As with picoArray, memory in XS1 is provided in small per core memories (between 64 KB and 256 KB per core depending upon the device). Unlike picoArray any core may access any other core's memory over the interconnect. However for both picoArray and XS1 there is no concept of caching from a larger external memory store. This could be implemented but it would be done via software accessing an external memory as a peripheral, with all caching implemented in software.

2.6 Summary

Multi-core architecture is very diverse, there are a wide range of plausible core configurations, ways to interconnect cores and methods to use this interconnect. Whatever design is chosen it is vital that it allows effective use of the provided resources. The design of the SCC, for example, is problematic because of its retrofitting of message passing capabilities into an existing cache architecture. The Tile Architecture as a counterexample offers great flexibility in how interconnect can be used allowing a wide range of programming styles. Multi-core architecture is also proving useful in more application specific areas, the XS1 and picoArray use it to great effect to allow the implementation of applications that traditional were confined to the realm of ASICs and FPGAs.

It seems certain that silicon process technology will continue to allow core counts to increase, with increasingly sophisticated interconnection mechanisms being utilised to enable core to core communication. This interconnect must be utilised effectively both by the architecture and the programmer to enable good scalability.

In this chapter current technology trends and how they motivate the creation of multi-core architectures has been reviewed. This leads into a discussion of architectural communication and how architectural mechanisms can be used from within software. Several examples of multi-core architectures are the presented. In the next chapter the idea of fine-grained architecture is discussed in more detail which motivates the creation of the Mamba architecture.

Chapter 3

Fine-grained computation

3.1 Amdahl's Law

No discussion of speed-up in parallel computation would be complete without reference to Amdahl's Law [5] which can be seen in equation (3.1), it relates the total speedup S that can achieved when parallelising a fraction f_p of a program across n cores.

$$S = \frac{1}{(1 \quad f_p) + \frac{f_p}{n}} \tag{3.1}$$

A major result of Amdahl's law is that the maximum speedup that can be achieved by a program running over an increasing number of cores is limited by the fraction of it that is parallelisable f_p as shown in equation (3.2)

$$\lim_{n \to \infty} S = \frac{1}{1 \quad f_p} \tag{3.2}$$

So if we wish to produce software that scales well with increasing core count we must seek to make f_p as large as possible. This is the principle behind fine-grained techniques, by providing the programmer with tools that encourage them to split a program into as many small parallel tasks as possible f_p can be maximised, thus achieving good scalability.

An issue with Amdahl's law is the simplicity of the model. Effectively it assumes that there is a sequential phase followed by a parallel phase, the sequential phase may only run on a single core and the parallel phase may run across any number of cores. Whilst some programs may approach this style of execution, especially if you consider having multiples rounds of the sequential phase followed by the parallel phase it is clearly not a realistic model of all programs. Crucially it does take into account any synchronisation or intercommunication between the parallel threads of execution, in these 'critical sections' a thread may have to wait for another thread to complete some task before it can proceed (for example a thread may have to wait for another thread to finish modifying a data structure before it can modify it).

Eyerman and Eeckhout have extended Amdahl's law to take into account critical sections [31]. They extend the model to include a contention probability P_{ctn}, the probability that two critical sections contend, that is one must wait to the other due to parallel access of some shared resource. The parallelisable fraction f_p is split into two parts $f_{p,ncs}$, the part of parallelisable fraction that doesn't require access to critical sections and $f_{p,cs}$, the parallelisable fraction that does (i.e. it may have to wait for some other thread when in this fraction), $f_{p,ncs} + f_{p,cs} = f_p$. They derive equation (3.3), where f_s is the sequential fraction of the program i.e. $f_s = 1 \quad f_p$

$$\lim_{n \to \infty} S = \frac{1}{f_s + f_{p,cs} P_{ctn}} \tag{3.3}$$

This shows that the speedup that can be achieved when scaling a program across an increasing number of cores is not only limited by the size of the sequential fraction, but the time it spends in critical sections. Whilst this result is relatively obvious, Eyerman and Eeckhout place it on an analytical basis. The effect on a fine-grained system is that not only must a programmer be encouraged to split a program into as many sub-tasks as possible to increase f_p but that P_{ctn}, the probability of contending, and $f_{p,cs}$, the fraction of time spent in a potentially contending situation, must also be reduced. This is partially up to the programmer but synchronisation between threads of execution is a necessity so they can only do so much. The programmer must be provided with synchronisation techniques that carry a low overhead reducing $f_{p,cs}$ and encouraging the programmer to use them more freely at a finer-grained level, reducing P_{ctn} as well.

This new version of Amdahl's law is based around the idea of synchronising using critical sections, where a lock protects some shared resource, it doesn't consider synchronising by other methods (for example synchronising by communicating over FIFO channels). However a critical section is modeled by a fraction of the program that might contend and when it does it must wait for another thread (serialising execution), such a general description also applies to other forms of synchronisation. For example if using FIFO based communication a consumer thread may have to wait for a producer thread if a FIFO it wishes to consume from is empty. This is much like the thread having to wait for another thread to release a lock before it can continue and could be modeled very similarly with a probability that the thread will have to wait for certain fractions of the execution time similar to the contention probability P_{ctn} and the fraction $f_{p,cs}$ spent in critical sections.

3.2 Fine-grained software

There are many approaches to a purely software based fine-grained model (that is one that, whilst requiring a multi-core or multi-threaded architecture so it can execute multiples threads in parallel, the architecture does not need to specifically support fine-grained mechanisms). Crucially any software based model needs to provide low overhead synchronisation and scheduling on top of available primitives. Typically these would be provided by an operating system and may require expensive system calls to use. A software approached to fine-grained computation may use these as a base for lighter user-mode based primitives (for example it may create one OS thread per core but then implement its own lightweight task switching inside each thread).

3.2.1 Message passing

One approach to building software in a fine-grained manner is message passing, a program is broken into isolated parts that can communicate only by sending messages (there is no shared memory), this approach has been formalised in the actor model [42]. By enforcing isolation between tasks there is no possibility over contention for shared resources, though a task may be forced to wait for a message to arrive. By dividing the program into more tasks the probability that there are not enough tasks to use all available cores can be lowered, so it is vital that a low overhead message passing and task scheduling system is provided.

Erlang [8] is a language designed around message passing, it provides a very lightweight threading implementation which uses user-space threads implemented on top of OS provided threads so a programmer is able to create as many as they need rather than needing to find an optimal number. This is combined with a user-space message passing implementation. As well as enabling fine-grained concurrency Erlang also works well for distributed systems, systems where multiple physically distinct process nodes are connected via some network.

3.2.2 Lightweight threads

If a program is to be split into many smaller tasks to be executed concurrently, there must be a way to execute these without the overhead of scheduling these tasks dominating execution time, effecting scalability. Whilst an operating system will provide threading primitives they are heavyweight objects, carrying much state with them that is not necessary for small tasks. One approach is to use work queues with work stealing [13]. A certain number of threads are created (matching the number of cores available for example) each with a queue of work or tasks to be completed. A thread proceeds working its way through the queue completing tasks, which may in turn queue more tasks to be completed. If a thread runs out of tasks to complete it may steal from another thread's queue, ensuring that all threads are kept busy. Provided the number of tasks relative to the number of threads is kept high and they do not contend much scalable concurrent software can be created. A task can be represented in a very lightweight manner compared to an operating system thread and there is no context switching overhead on finishing one and starting another so the number of tasks can be large.

Intel's Threading Building Blocks (TBB) library [47] is an example of a fine-grained task based system that utilises work stealing. A task is represented by an object with an execute method, a scheduler is responsible for distributing tasks between processors and executing them. In the course of executing a task can spawn further tasks forming a hierarchical relationship between them. Contreras and Martonosi performed a study on the overheads of TBB [20], they found that the overhead of the library effected scaling at higher core counts, in some cases an overhead of 3% with 16 cores (allowing a 14.8 speedup) grew to an overhead of 52% with 32 cores (allowing a 14.5 speedup). Excessive task creation was also problematic, one benchmark when altered to spawn 6M tasks to expose further parallelism instead of 6K tasks went from a 19 speedup to 10 . The two major components of the overhead was the library waiting for resources to become available and overhead caused by atomic operations used for synchronisation.

3.2.3 Software transactional memory

Locks are an effective and simple way to ensure safe access to a shared resource. However using them can drastically increase software complexity, especially as the number of locks used increases as it would when using locks in a fine-grained manner. Transactional memory is a technique that allows software to access a shared area of memory safely without needing locks. The code accessing the shared region needs to declare that it must be executed as a transaction and the transactional memory system can ensure it executes safely.

This way of dealing with shared resources matches well with a fine-grained style. A programmer breaks their software into many small tasks that may access shared memory as they wish in small transactions exposing a large amount of parallelism. The system then enforces safety [41]. Transactional memory accomplishes this by logging reads and writes that occur in the transactions and only allowing the writes to commit (i.e. become visible to other threads) at the end of the transaction if the logged reads and writes do not conflict with any other transactions.

A purely software based approach to transactional memory (known as STM, software transactional memory) is possible and many implementation exist [38, 25, 104]. A software approach requires that some (but not all, only those relevant in deciding whether or not a transaction can commit) must be instrumented, this could be done by hand or by a compiler. The problem is this instrumentation adds a large amount of overhead. Cascaval et al. [15] examined this overhead comparing several popular STM implementation finding that the overhead is such that in certain cases the transactional version of a benchmark did not achieve the performance of the non-transactional benchmark on any STM implementation even when running over eight threads. They explore the overhead in detail and suggest some possible solutions but conclude that lowering the overhead to the point STM systems become usable is very challenging.

3.3 Fine-grained hardware

Fine grained hardware could be split into two separate categories, that which accelerates or supplants some of the software techniques discussed above and that which uses new models of execution that require large amounts of explicit parallelism. Both categories will require software to be rewritten to take advantage of their capabilities but the first category will often take the form of extra functionality that can be added to existing core designs whilst the second may require entirely new core design or the use of a co-processor.

3.3.1 SIMD

Single Instruction Multiple Data (SIMD) is one of the classes of computer architecture in Flynn's taxonomy [32]. In a SIMD architecture a single instruction stream is applied to multiple data streams. A multi-core system would be described as MIMD, Multiple Instruction Multiple Data, each instruction stream (i.e. thread) has its own separate data. Single instruction single data, SISD would be a single core system and Multiple instruction single data, MISD, systems are rarely encountered.

SIMD techniques can be very effective methods of fine-grained computation for certain applications. They can achieve great parallelism at low overhead. This is achieved by using a single instruction stream over multiple data streams. The overhead of fetching and decoding instructions can be amortized by the multiple data streams that are operated on in parallel. Anything involving a large degree of data parallelism should map well to them. A notable SIMD implementation is Intel's SSE and AVX instruction set[48]. The SSE instruction set (there are multiple revisions, AVX is the latest of these) introduces wide vector registers that can hold multiple floating point numbers at once, special instructions operate upon them performing arithmetical and logical operations on each of the floating point numbers at once (effectively the registers and instructions are like a separate co-processor).

Another notable SIMD architecture is that of graphics processing units (GPUs). Originally designed as special purpose accelerators for computer graphics the expanding feature set and increase programmability of the graphics pipeline designed to exploit the highly data parallel nature of rendering has lent itself well to more general purpose tasks leading to the rise of general purpose GPU (GPGPU) techniques [51].

The rise of GPGPU has led to NVidia producing GPUs targeted at high performance computing, an architecture known as Tesla [66]. Telsa supports a very high number of threads in hardware, supporting lightweight thread creation and zero-overhead scheduling. It groups threads together into objects known as warps, which are the unit of scheduling. When executing a warp an instruction is chosen to execute which is broadcast to all threads. Each thread in a warp may branch independently but when this happens not all threads will be executing the same instruction, so first the instructions for the threads that have followed the branch will be broadcast, and the threads that did not follow the branch will simply ignore the instructions and idle, before the threads that did take the branch finish the branch and all threads will execute the same instruction together once again. This means that any code that relies on a high degree of branching where each thread may take different branches will map poorly to the architecture. Zero-overhead scheduling is achieved by having multiple warps available and dynamically choosing which to run at each clock cycle.

Some studies have achieved speed-ups of up to two or three orders of magnitude using GPGPU hardware compared to CPU [92, 99] with others reporting more modest but still impressive speed-ups of up to 40 [35]. These kinds of results have caused many to turn to GPGPU as the solution for gaining greater performance. Lee et al. [62] studied optimising a variety of kernels for both CPU and GPU and found that whilst well optimised GPU implementations did out perform CPU implementations the speed-up was far more modest around 2.5 . One explanation for the great difference in speed-ups observed could be that when writing software for a system that doesn't inherently require fine-grained computation a programmer doesn't attempt to extract the full amount of parallelism because it isn't needed to get the software working. When writing for a GPU the architecture forces the programmer into a fine-grained way of thinking, causing more parallelism to be exposed in the code which can then be exploited.

3.3.2 Hardware threading

When exploiting fine-grained computation in software one of the critical issues was scheduling, the overhead of the OS scheduler combined with a context switch is too high to allow many fine-grained threads, lightweight task based systems were built on top of heavyweight OS threads to combat this. Another approach is to add support for such lightweight task based systems into hardware. Kumar et al. have proposed Carbon [57]. It augments each core with a local task unit (LTU) and introduces a global task unit (GTU) used by all cores. The LTU and the GTU work together with ISA extensions to implemented hardware task queues. A thread may enqueue a task, which will be sent to the global task unit, LTUs prefetch tasks from the GTU, so when a thread on a core is done with a task it may dequeue the next one from the GTU. The GTU is responsible for scheduling tasks, choosing which task goes to which core, which is does via a work-stealing algorithm. Carbon is found to give a large performance benefit over similar software based systems and come close to an ideal hardware scheduler (one which can enqueue and dequeue tasks with no latency).

An evolution of Carbon was proposed by Sanchez et al. [87]. They observed that Carbon imposes a single scheduling algorithm that may not always be optimal for a given application. Furthermore Carbon introduces a number of hardware structures that serve the single purpose of hardware scheduling. They proposed a hybrid scheme, using asynchronous direct messages (ADM, discussed in the background chapter) that provide direct exchange of small messages between threads in hardware. ADM is then used to implement a family of software schedulers. Threads were split into workers and managers, a worker actually executes the program, enqueuing and dequeue tasks from a thread-local software queue. The managers dealt with distributing and balancing tasks and to support greater scaling were made into a hierarchy with higher level managers coordinating lower level managers. When an ADM scheduler was tailored to the application ADM based scheduling could outperform Carbon by up to 70%.

Another possibility for hardware threading is to support executing multiple threads at once on the same core. Each thread has associated hardware state (registers etc.) and a core has multiple copies of this state. So it can make a decision on which thread to run each cycle. The Cray XMT [75] is one system that implements this and supports 128 threads per core. In order to simplify the pipeline it enforces that each thread may only have a single instruction in the pipeline at once. So to get full performance out of the system many threads are required and the system relies on this to help hide latency. The UltraSPARC T2 [90], discussed in the background chapter, is another system implementing hardware threading in the same manner, which supports up to 8 threads per core. Unlike the XMT it allows multiple instructions from the same thread to be in the pipeline at once, giving better performance at low thread counts.

The Anaconda architecture proposed by Moore [77], implements a sophisticated hardware scheduler based around a priority queue which allows a hardware implementation of earliest deadline first and fixed priority scheduling, which makes the scheduling system suitable for real-time applications. The key to this scheduling mechanism is a hardware sorter that can efficiently insert and extract from the hardware priority queue. This scheduling mechanism is combined with the concept of data driven microthreads. Small threads that begin execution when the data they need is ready, allowing a coarse grained data-flow style of execution. A microthread is created by an activation frame which is a structure held in memory (though usually will be found within a local cache) that gives up to 16 input parameters for the microthread. Each input parameter has a matching presence bit (stored as a 16-bit word at the beginning of the activation frame) when all presence bits are set it is sent to the scheduler (a special store instruction is used to store to the activation frame). Part of the activation frame gives a deadline or a priority that is utilised by the scheduler.

3.3.3 Hardware transactional memory

Hardware transactional memory (HTM) is identical to STM (discussed above) in intent but implemented architecturally. The large advantage of an HTM system is they can have far less overhead than an STM system. A major source of overhead in STM is the need to instrument memory accesses in HTM, as the hardware must execute the memory accesses extra, instrumentation isn't required.

HTM implementations often use the cache and its coherency mechanisms as a way to track memory accesses and ensure transactions do not conflict as was done in the original HTM implementation by Herlihy et al. [41]. This has the advantage of simplicity and allows HTM to be implemented on an existing architecture without needing too many extra features to be added but fails when a transaction overflows the cache or is long-running. Damron et al [23] suggested the use of this style of HTM with a STM fallback. There also exist unbounded HTM systems that effectively implement STM systems fully in hardware [6] however these have a high degree of complexity. Sun created a commercial implementation of HTM in the Rock processor [17] however the development of it was cancelled. Intel have recently revealed the Transactional Synchronization Extensions (TSX), which will be available in the Haswell processors [49] which provided HTM support but based around detecting conflicts between cache line accesses, with the limits that implies.

HTM could enable the fine-grained possibilities transactional memory promises without the overheads of STM. However current approaches are limited or too complex, so it has seen little take-up in commercial processor implementations. With Intel's inclusion of TSX in its new architecture transactional techniques may become more popular but due to TSX's limitations could not replace more traditional synchronisation entirely.

3.3.4 Hardware synchronization

Specialised hardware can be introduced that supports low overhead fine-grained synchronization. Zhu et al. [107] proposed the synchronization state buffer. Based on the idea that at any given instant only a small fraction of memory locations are currently participating in synchronization a small buffer (the synchronization state buffer or SSB) is attached to the controller of each memory bank. This buffer manages the state of actively synchronizing memory, it consists of a number of entries associated with an address. Each entry manages the synchronization state of its associated address via a variety of locking methods. New instructions are introduced that can interact with the SSB allowing the creation of locks associated with a particular address. An entry is only required whilst at least one thread is using the lock so provided the locks are fine-grained and the critical sections short the small size of the SSB should not be an issue. Still a software fallback mechanism is required for the cases where the buffer does overflow. SSB is demonstrated to scale well as well having a small overhead compared to other lock implementations (using test and set or compare and swap).

Ributzka et al. [85] extended the SSB concept (to produce the E-SSB) to allow a non-strict approach to fine-grained synchronization where a consumer can read a value before a producer has produced it. The consumer will only have to wait when it actually uses the value if the producer has yet to produce it rather than stall at the point of reading the value waiting for it to appear. When a store (a special store instruction is introduced for using the E-SSB) is executed an entry in the E-SSB is created showing it has occurred. A matching load looks for this entry if it finds it the data is returned and the thread continues as normal. If the load occurs before the store no entry is found, so no data is returned but the thread requesting the load and the register destination is noted. Scoreboarding is used to allow the thread to continue until it actually uses the loaded value. In the meantime if the store has occurred then the data will have been returned and the thread will not stall, otherwise it will wait until the store happens. The implementation of the E-SSB achieved very good scalability and outperformed other synchronization constructs.

Tullsen et al. [100] examined the basic mechanism of a spin-lock, when a thread repeatedly checks the value of a word. A particular value indicates that the spin-lock has been acquired by another thread and another value indicates that nothing has acquired the spin-lock. By using an atomic operations such as a compare and swap (CAS) a thread can observe the spin-lock is not acquired and acquire it. If it fails to acquire it it tries again. The main issue is the repeated spinning, whilst a thread is waiting on the spin-lock it is wasting execution resources as well as generating needless memory accesses, potentially competing with another thread for access to the cache. This spinning aspect was removed by introducing the lock-box. A processor core has one lock-box per hardware thread which contains the address of a lock, a pointer to an instruction and a valid bit. When a thread wishes to acquire a spin lock it uses a specialised acquire instruction. This proceeds to acquire the spin-lock as usual but if it fails instead of spinning it writes the address of the lock and the address of the just executing acquire instruction to the lock box and the thread is paused from execution. When another thread releases a spin-lock with a specialised release instruction the address of the released lock is checked against all of the valid lock-box entries, if any matching entries are found the corresponding threads are woken up.

The lock-box scheme was evaluated using a SPEC benchmark, espresso, and some livermore loops. All of these contains loops that can be parallelized but only if the synchronisation used has a small overhead. They found the lock-box scheme allowed parallel speed-ups on four out of five of the loops examined where no speed-up would be possible on a more conventional multiporcessor. The lock-box scheme also has the advantage of being straight-forward and doesn't need massive architectural change. However it relies on all lock-boxes being made aware of any lock releases that effect them. This presents a challenge to scaling, but it is a very similar problem to the one faced by any cache coherence mechanism so any large multi-core system that had a cache coherence system could likely use it to implement a scalable lock-box scheme.

Kaği et al. [52] looked at a variety of spin-lock based synchronisation mechanisms and found the most effective of these to be queue on lock bit (QOLB). QOLB, like Tullsen's lock-box aims to avoid the needless work created by a thread that is spinning waiting to acquire a lock. When a processor attempts to acquire a spin-lock it receives a shadow copy of the cache line that holds the lock, it spins on this shadow copy, avoiding unnecessary memory accesses and network traffic. A hardware queue of waiting processors is maintained by holding queue entries in the cache line. When a processor releases a lock it can give it to the next processor in the queue. Kaği et al. concluded that QOLB has consistent and large performance gains compared to all of the other locking techniques they studied. Whilst QOLB is clearly an effective mechanism it does rely on the in cache queue entries. If you wish to allow many threads on a single processor the basic QOLB mechanism will not be effective. If several threads on the same processor are attempting to acquire the lock they will still interfere with each other and other threads on the same processor. The single queue entry in the lock's cache line can only point to a single other processor, it isn't sufficient for storing a queue where multiple threads on a single processor may be in it.

The Stanford DASH Multiprocessor [64] which is an multiprocessor architecture that utilises directory based cache coherency contained a specific optimisation for locking. The directory held information on what processors were spinning on a lock. When the lock was held each processor would be spinning on a local copy of the lock, the issue is when the lock is released all copies of the lock would be invalidated and re read, causing much needless network traffic. Instead a single waiting cluster of processors can be chosen at random from the knowledge in the directory of what is spinning on a particular lock, only that cluster needs to be informed of the lock release avoiding needless network traffic.

A common theme amongst the variety of hardware based synchronization mechanisms is the idea of extending or augmenting the set of memory operations. Solomatnikov [94] proposed a general scheme called Smart Memories. Where by introducing a programmable aspect into the operation of on-chip memory a variety of different synchronisation and communication mechanisms can be employed. The Smart Memories architecture divides up a multiprocessor into a number of quads, each of which contained four tiles with a single protocol controller that can execute basic memory system operations and coordinates memory accesses inside the quad. Each tile consists of two cores and a configurable memory. The configurable memory consists of an array of data words and associated metadata bits. Programmable logic is used to modify precisely how this memory array is accessed, the metadata bits are fed into the logic as well as updatable by it (e.g. you could program the configurable memory to disallow writes to all data words with a certain metadata bit set). More specialised comparator and pointer logic allows the use of the memory array as tag storage for a cache or as a FIFO.

The Smart Memories Architecture was used to implement shared, streaming and transactional memory models. The shared model is a implementation of a standard broadcast based MESI cache coherence protocol. The streaming model is one where cache coherency is not used, software managed local memories along with DMA are utilised instead. The transactional memory model implements a transactional memory system with a hybrid model, hardware provided primitives and used by software to build a full transactional memory system. In each case the Smart Memories Architecture can be used to implement an effective example of the model and due to the flexible nature of the system the models can be easily tuned to improve them without the need for architectural changes. A direct comparison was done between the streaming and shared memory models for a range of applications, some applications performed better with a streaming memory model, others with a shared memory model. The advantage of smart memories is both can be used to their best effect.

In the shared memory model the metadata bits were utilised to give fast fine-grained synchronisation using the concept of full / empty bits discussed below. New load and store instructions allowed a processor to be stalled waiting for a word to become full or empty. The implementation of this was tested by recompiling one of the benchmarks (a hypersonic flow simulation) to use the new instructions for access to shared data structures. The benchmark is unusual in that whilst locks should be used to access these structures they aren't necessary. This is because it involves a statistical averaging of many steps the data races that occur without locks don't significantly effect results. The recompiled version using full / empty bits did not differ significantly in performance from the version which had no locking.

Whilst the Smart Memories Architecture is extremely flexible that flexibility incurs a cost in both extra area required for the smart memory implementation but more crucially in complexity. The advantages of being able to implement a wide variety of memory and synchronisation models to suit your specific application are obvious but offering such a large range of functionality can lead to complex programming environments and intractable debugging issues. These can be lessened by having an operation system use the general functional to provide a restricted set of well defined primitives, but if they suffice a simpler, smaller, architecture implementing precisely those primitives may have been a better option.

3.3.5 Full / empty bits

The concept of a full / empty bit is an old one. An early example of their use can be found in the HEP system [93], every word in memory in the HEP had an associated full / empty bit. When loading a value a load instruction could wait for the bit to become full or when storing the store instruction could wait for a bit to become empty. This allows fine-grained synchronisation at a word level without any limits imposed by hardware structures other than the ultimate limit of available memory (The E-SSB described above enables the same style of computation but with a limit imposed by the size of the hardware buffer used).

A far more recent architecture that utilises full / empty bits is the Cray XMT [75]. The XMTs support of full / empty bits is vital for its threading model which relies on a high degree of parallelism. A particular thread can spin (continuously polling until it sees the value it wants) on a full /empty bit, trapping to software after a while to allow the operating system to deschedule it.

Li et al. recently introduced the Lightweight Chip Multi-Threading (LCMT) architecture [91]. It utilises hardware scheduling with a tagged memory. Threads have an in-memory representation consisting of their current register contents that can be swapped in and out of a register file. Every word is tagged with an extension bit (xbit), which serves a similar purpose to full / empty bits. A thread reading a word with an unset xbit can stall and be descheduled, waiting for the word to be written. However only a single thread can stall waiting for a word to be written. In the case of multiple threads reading a word with an unset xbit the LCMT architecture traps to a software handler.

The Godson-T [26] is a recent multi-core architecture, based on MIPS. It offers full / empty bit based synchronisation. However instead of tagging all of memory with full / empty bits they are only included within the cache, so synchronisation using them must happen on words currently in the cache. Any thread waiting for a location to become full must spin waiting for the bit to become set, there is no direct interaction with a hardware scheduler.

Full / empty bits are utilised in the Mamba architecture, but throughout this dissertation they are referred to as presence bits.

3.4 Summary

In order to achieve scalability in the face of increasing core counts Amdahl's law tells us we must maximize available parallelism. Fine grained techniques are an ideal way to do this by encouraging a programmer to expose as much parallelism as possible. However the scalability of a fine-grained system is limited by the overheads of the scheduling and synchronisation system used. It is vital that these are kept as low as possible to allow good scalability. One way of achieving the low overhead needed is to implement suitable primitives in hardware, however if this involves a limited hardware resource this places limits on how the primitives can be used. Whilst software fallback is possible a balance needs to be struck between excessive consumption of resources and limiting performance due too much reliance on the fallback. A software fallback system can also complicate programming, separate code paths may need to be provided to deal with the cases hardware cannot. On the other hand full hardware implementation may add great complexity to the design as demonstrated by boundless HTM implementations. Ideally a hardware provided mechanism should be simple enough that full limitless implementation does not great complexity but also should be flexible enough to deal with a large range of possible synchronisation scenarios.

In this chapter the overheards of synchronisation and scheduling are presented as the ultimate barrier to software scaling in multi-threaded and multi-core systems. A variety of hardware and software systems for reducing this overhead to allow the creation of fine-grained systems are discussed. In the next chapter Mamba is presented, a new fine-grained architecture designed for producing highly scalable fine-grained software.

Chapter 4

Mamba architecture

4.1 Overview

This chapter presents the Mamba architecture. It is a multi-core architecture aimed at the challenges of the multi-core era. Mamba offers a lightweight hardware managed threading model to allow the use of fine-grained techniques encouraging software that scales gracefully as the number of cores available with each silicon technology generation increases.

Mamba uses presence bits. Each word in memory has an extra presence bit that indicates whether or not data is present at that location. A thread that loads an empty word can be descheduled, only becoming ready again once the non-present word it attempted to load becomes present. This is a simple mechanism, not requiring much extra complexity in hardware but can be used for a wide range of synchronisation tasks (see the software techniques chapter). A key part of the architecture that makes presence bits effective is the hardware managed ready queue used by the scheduler, this is backed by main memory so it doesn't impose any artificial limits on software.

Mamba is based on ideas from an earlier architecture, Anaconda, proposed by Moore [77] and discussed in the Fine-Grained Computation chapter. Much like Anaconda Mamba represents a thread using an activation frame, an in-memory representation of a thread. Activation frames are written to with a special store instruction, when all presence bits on a activation frame are set that may be scheduled by the hardware scheduler. The major differences between Mamba and Anaconda lie in the scheduling system and usage of presence bits. Anaconda aims to support real-time applications so requires a sophisticated scheduler utilising a priority queue to support earliest deadline first and fixed priority scheduling. Mamba does not aim to support real-time applications so implements a simple FIFO scheduling scheme. Anaconda utilises the idea of microthreads, a thread which does a small amount of work before spawning further threads with its results, giving a coarse-grained data flow method of execution. Whilst Mamba is capable of this it is expected that many threads in the system may be long lived, though may spend large periods descheduled waiting for an empty word to become present. Mamba also utilises presence bits over the entirety of memory, in Anaconda they are only used within activation frames.

Mamba has been implemented on FPGA, using high speed serial links to connect multiple FPGAs together to simulate a single multi-core processor. Each Mamba core is a simple, scalar, in-order core supporting fine-grained multithreading allowing it to run 8 threads at once. Mamba is based on the MIPS64 ISA [74] that has been extended to support Mamba's threading, communication and synchronisation features. The evaluation of Mamba compares it to a MIPS64 implementation that lacks these features but is otherwise identical. The aim of the evaluation is to demonstrate the utility of these features in enabling good software scaling and ease of fine-grained concurrency (which is the thesis of this dissertation).

Implementing the architecture on FPGA as opposed to constructing a simulation of it in software was chosen for two reasons. An FPGA implementation, even if it is two orders of magnitude slower than an ASIC implementation is still fast enough to actually run benchmarks with full dataset sizes. A software simulation, especially as the core count increases severely limits the evaluation possibilities. An FPGA implementation also provides strong evidence that the architecture can be realised.

4.1.1 Interconnect

Mamba uses a 2D mesh topology to connect cores. Two separate networks are used, one for requests and another for responses to avoid protocol deadlock. Both networks are otherwise identical, dimension ordered routing is used with on-off flow control between routers on an FPGA. Connections between FPGAs are accomplished with high-speed serial links. This setup has two major advantages. Firstly it is flexible, further cores can be simply added into the mesh with no changes needed due to the simplicity of the topology and the routing. Secondly it is ordered, messages between two cores will be received in the order they were sent, whilst other topologies and routings may be able to better deal with congestion this can disrupt message order which has an impact on the memory consistency model.

4.1.2 Architectural communication

Mamba does not directly expose the interconnection network to software. It presents a single shared memory space to all cores but it doesn't implement a cache coherency protocol or more accurately it implements a null cache coherency protocol. Caches are always coherent without the need for a protocol. This is accomplished by only allowing a cache line in memory to be cached in one particular place (that cache owns that line), any core accessing memory in a line that its cache doesn't own must send that memory request to the remote cache that does own the line where it will be serviced. Memory is thus divided into a local area and a remote area with a trivial mapping between an address and the core that owns that address. This is a hardware implementation of a PGAS style programming model and gives software direct control over all communication that occurs, a local access goes to the local cache, a remote access goes to the remote cache (with communication cost directly proportional to the distance between the local and remote cores). Direct access to the interconnection network gives similar control over communication however it allows the possibility of deadlock that must be dealt with in software.

Another advantage of the simplistic coherency model combined with the in-order delivery of the network is consistency. As all messages between a core and a cache will be received in the same order they were sent and the cache processes all requests in a sequential order sequential consistency is maintained in each core's memory area, sequential consistency is not maintained between memory locations that are placed in two different places.

A key part of any communication is the notification that communication has occurred, not all communication may require data, but all communication does require some indication that it has actually happened. Mamba supports this via the concept of presence bits, also known as full/empty bits. Every 64-bit word in memory has an extra bit. When this bit is set the word is said to be present, writing to a word sets that word's presence bit. There is a mechanism that allows a thread to block on loading a word that is non present, with the thread only being woken when another thread writes to the word setting the presence bit (this mechanism is discussed in detail in the this chapter). This provides a primitive synchronisation mechanism allowing fine grained notification, locking and the ability to exploit data parallelism.

4.1.3 Software

Each Mamba core supports up to 8 active threads in hardware and has a hardware scheduler. A thread is represented in memory by an activation frame (AF) and the hardware scheduler of a core will run any ready AF that is placed in the core's local memory, this makes it cheap to create new threads. Combined with the ability to wait on a non-present word and the efficient hardware context switch efficient fine-grained programming becomes possible. This model also scales well, splitting the same amount of work up between more threads doesn't necessarily have much of a performance impact. Encouraging software to split computation into many threads like this leads to software that scales with an increasing number of cores without the need to revise the software.

Presence bits provide a general mechanism for thread notification. A thread is able to wait and be descheduled until it is signaled by the word it is waiting on being written to. As a thread does not need to invoke an operating system to be descheduled pending some event notification has a very low overhead. This allows simple, flexible fine-grained synchronisation and easy exploitation of data parallelism. A structure called a notify chain (described in the chapter on software techniques) is proposed to deal with general notification.

4.2 The programming model

Mamba is programmed with a MIPS64 based ISA, only the user mode component of MIPS64 is included. The programming model has two major features that are not part of MIPS64.

Hardware Threading Mamba supports a number of hardware threads limited only by the size of memory available. Threads are represented in memory by an activation frame (AF) which holds the current register values for the thread, the program counter and an exception status word (ESW). This is all of the state required by a thread.

Presence Bits Every 64-bit word in memory has an associated presence bit. This is used as a fine-grained synchronisation and notification primitive. A thread which attempts a load of a non-present word will stall until it becomes present. A hardware scheduler deals with descheduling threads when they are waiting for a word to become present and scheduling them again when it does become present.

A Mamba system consists of a number of nodes connected in a grid network. Each node has a single processor core that executes up to 8 threads at once and is assigned an area of the global address space, referred to as the node's local address space (or the space the node owns). Any time a thread executes a load or store instruction if it targets a non-local address a message will be sent across the network to the node that owns that address and the memory access will be performed via its cache. Any accesses to the local address space will be done via local cache with no messages sent across the network. As a result remote accesses will be slower than local accesses, but a programmer can precisely control when communication occurs across the network by choosing where to place their data.

4.2.1 Activation frames

An activation frame (AF) is a representation of a thread and is a 32x8 byte block of memory. This has the current register values, program counter and exception status word (ESW), explained in further detail below, of the thread (see figure 4.1). An AF that has all of its presence bits set is a ready AF, one that represents a thread that can be run. It will be placed in the ready queue, a structure managed by a hardware scheduler. A thread, represented by an AF, is run in a context, a hardware resource, there are 8 per core so up to 8 threads may be running in a core at any time. An AF is placed in some node's local address space, the thread it represents will only run at that node. The hardware scheduler deals with scheduling and descheduling threads as appropriate.

To create an AF the contents of it are written to a 256-byte aligned blocked of memory in the local space of the node that has been chosen to run the thread it represents. A special store instruction **SDA** exists to write words to a new AF. It signals to the processor that after doing the store it should check the presence bits of all words within the AF and add it to the scheduler's ready queue if they are all set. When a thread is done it can destroy the AF (that is ensure it will no longer be scheduled and allow the memory to be reused) by executing an instruction that branches to itself.

As each register in a thread is represented by a word in its AF every register is effectively addressable. The register file can be seen as a specialised cache that holds the contents of the activation frame whilst the thread is running. A load or store targeted at an AF whilst it is currently occupying a context (i.e. the thread it represents is being executed)

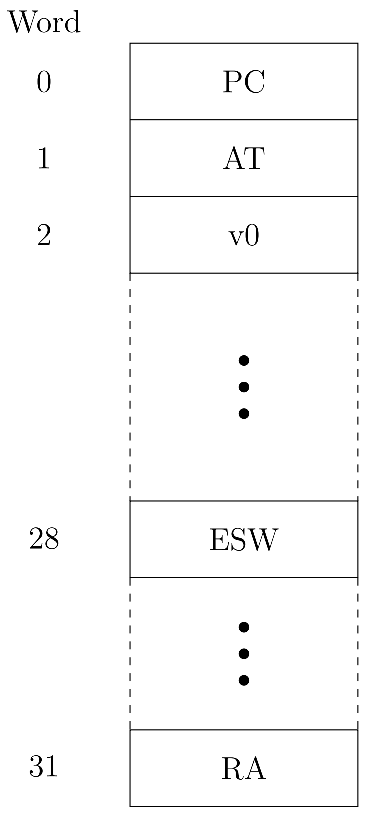


Figure 4.1: An activation frame, PC is the program counter, ESW is the exception status word, AT, v0 and RA are specific MIPS registers (which is what fills the unlabeled slots)

may interfere with the execution of that thread, potentially causing it to stall forever. Loading or storing to an AF that doesn't currently occupy a context but is either in the ready queue or waiting for a word to become present may also interfere with the execution of that thread. Because of this between the point the AF is created with **SDA** and the point it destroys itself by an instruction that branches to itself the result of any memory access to that AF is undefined.

4.2.2 Presence bits

Every load or store interacts with a presence bit. There is one presence bit per 64-bit word, an access less than 8 bytes to a non-present word is considered an invalid operation, further details are given below. A load of a present word acts normally, as soon the data is read from cache it is returned to the thread executing the load so it can use it immediately. A load of a non-present word causes the thread that triggered the load to be stalled, but only at the point it attempts to use the data. A store to a non-present word sets the presence bit as well as writing the data and waking up any thread that was waiting for the word to become present.

As well as every 64-bit word, every register has an associated presence bit. When a thread uses a register that is non present it will cease executing until it becomes present. The register presence bits are taken from the presence bits of the corresponding words in the

AF. So when an AF is first created and the thread it represents runs all registers will be present. When a load instruction is executed the destination register for the load becomes non-present. When the load has completed the result will be written into the destination register and it will become present again. A load in Mamba is non-blocking so the thread will immediately execute the instruction following the word even if the load has not completed. In the meantime accessing the load's destination register will cause the thread to wait. This is the basic mechanism that allows a thread to wait upon a word becoming present, it will attempt to load the word to a certain register, when the thread tries to use that register it will pause, potentially becoming descheduled. When something else writes to the non present word the write will be forwarded to the currently non present destination register. This will make the register present again so the AF it is in will have all of is presence bits set so it becomes ready allowing it to be scheduled.

When a load occurs to a non-present word we want the response to that load to be generated when something writes to that non-present word. This is accomplished by the load writing the address of its destination register into the non present word (leaving the present bit unset). An address stored like this in a non-present word is known as a forwarding address. When a store occurs on that non-present word a read response is sent to that forwarding address. A special sentinel value (1, a non valid forwarding address as it's not 8-byte aligned) is used to indicate no forwarding address is present.

A problem occurs when a second thread attempts to load the same non-present word. A forwarding address is already written to it so it's unable to write its own (if it did then the first thread would never receive its read response and never get woken up). In this case a special response is sent back to the thread targeted at the destination register of the load. This special response is known as a read exception and will set a bit in the exception status word (ESW) corresponding to the register it's targeted at as well as setting the presence bit of the register (so the thread can continue to run). There are two styles of load with different exception behaviour. With **LD**, the normal load, a read exception being received will cause the thread to jump to an exception handler. With **LDNR** (load expecting no return) a read exception will only set a bit in the ESW, the software is responsible for checking the ESW after an **LDNR** to see if the load succeedethesis.pdfd or not. The instruction used to examine the ESW (which is a move from coprocessor instruction) specifies the particular bit that is wanted. The processor will pause the thread if it attempts to read the ESW whilst there is still an outstanding load (that is one that has yet to receive either a read or exception response) for the register that corresponds to the bit specified. So **LDNR** is used in situations where the software would expect there to be multiple threads attempting to access the same non-present word and has a mechanism to deal with that situation. **LD** is used where the software would not expect multiple threads to access the same non-present word. Exception handlers are only expected to be used to provide safety (if instead no new forwarding address were inserted and no response were sent on an **LD** to a location that already has a forwarding address then the thread would wait forever), not as part of a mechanism to deal with multiple consumers of a single non-present word. **LDNR** should be used in all cases where the software would expect any possibility of multiple consumers (and have a mechanism to deal with them, one such mechanism, notify chains are discussed in the software techniques chapter).

There is one more style of load, the vacating load **VL**. The vacating load clears the presence bit of the word it operates upon, if the presence bit is already unset then it returns a read exception, which like the **LDNR** instruction will just set the appropriate bit in the ESW but doesn't trigger an exception handler. The software is responsible for checking the ESW after an **VL** and taking appropriate action if it failed.

To summarise, loads and stores work as follows:

Load (Figure 4.2) Loads from a particular address (A) to a destination register with address (R). When a load is processed the presence bit of A is checked, if it is:

> Present — A read response is sent to the return address (R) with the contents of the word.
>
> Non Present — The contents of the word are checked, if it is a sentinel value, the return address of the load is written into the word and nothing else is done. If the sentinel value is not there (The sentinel value is a particular invalid return address), then some other load request has already written its return address into this word and an exception response is immediately sent. Upon receiving an exception response the bit corresponding to the destination register is set in the ESW, if the response was triggered by an **LD** an exception handler is triggered, if the response was triggered by an **LDNR** software is responsible for checking the ESW and taking appropriate action

Store (Figure 4.3) Stores data (D') to a particular address (A). When a store request is processed the presence bit of the corresponding word is checked, if it is:

> Present — The contents of the word are overwritten with the new data, nothing else is done.
>
> Non Present — The presence bit is set and the current word contents are checked. If the sentinel value is there, nothing further is done. Otherwise a load return address is there and a load response with the store data is generated to that address.

Vacating Load (Figure 4.4) As with the standard load above, the vactating load loads from a particular address (A) to a destination register with address (R) but it also clears the presence bit of the A when it does. When a vacating load is processed the presence bit of A is checked if it is:

> Present — A read response is sent to the return address (R) with the contents of the word and the presence bit of A is cleared
>
> Non Present — A read exception is sent to the return address (R), this will set the corresponding bit the ESW but it won't trigger the exception handler. Software must check the ESW after a vacating load and take appropriate action if it failed.

Mamba also introduces two instructions for manipulating presence bits, test and set **TAS** and test and clear **TAC**. Like a load these are given a memory address and a destination register. They will set or clear the presence bit respectively at the given address and return

Before Load		After Load		Response To Load	
1	**D**	**1**	**D**	**R**	**D**
0	**S**	**0**	**R**	None	
0	**F**	**0**	**F**	**R**	**Exception**

Figure 4.2: Detail of **LD** and **LDNR** operation. D is the data stored at the word being loaded, S is the sentinel value, R is the return address of the load and F is an already existing forwarding address. **0** refers to a non-present word, **1** refers to a present word

Before Store		After Store		Response To Store	
1	**D**	**1**	**D'**	None	
0	**S**	**1**	**D'**	None	
0	**R**	**1**	**D'**	**R**	**D'**

Figure 4.3: Detail of store operation. D is the data stored at the word being loaded, D' is the new data being stored, S is the sentinel value, R is a forwarding address. **0** refers to a non-present word, **1** refers to a present word

the state of the presence bit before it was altered to the destination register. When we **TAC** a present word as well as clearing the present bit we write in the sentinel value. When we **TAS** or **TAC** a non-present word we check what is stored there. **TAS**ing or **TAC**ing a non-present word that has a forwarding address is not allowed so if it is anything apart from the sentinel value a read exception is sent to the destination register of the **TAS** or **TAC**.

A final new instruction is **MYAF**, this is given a register number and computes the address of the register within the AF of the thread that executes the **MYAF**.

All of the new instructions introduced in this section along with the different loads and stores are summarised in table 4.1.

Before Load		After Load		Response To Load	
1	**D**	**0**	**S**	**R**	**D**
0	**X**	**0**	**X**	**R**	**Exception**

Figure 4.4: Detail of **VL** operation. D is the data stored at the word being loaded, S is the sentinel value, X is whatever is stored in the non-present word, R is the return address of the load and F is an already existing forwarding address. **0** refers to a non-present word, **1** refers to a present word

Mnemonic	Description
LD	Load instruction, clears presence bit of destination register when executed, it will result in a read response if the location being loaded is present, if not present a exception response will be generated if the non-present word already has a forwarding address. This exception will set a bit in the ESW corresponding to the destination register and trigger the exception handler.
LDNR	Load instruction identical to the above apart from a exception response will only set a bit in the ESW and won't trigger the exception handler
VL	Vacating load instruction, operates similarly to the other two load instructions apart from it clears the presence bit of the word it is loading. If the presence bit is already clear then an exception response will be sent which will set the appropriate bit in the ESW but doesn't trigger the exception handler.
SD	Store instruction, if the word being stored to isn't present then its contents will be checked. If it doesn't contain the sentinel value then a read response with the write data will be sent to the forwarding address stored in the non-present word.
SDA	Store to AF, works exactly as the store instruction above but in addition checks the presence bits of every word in the AF being stored to. If they are all set then the AF is ready and is placed on the scheduler's ready queue to be run.
TAS	Test and set. Sets the presence bit of a particular word and returns the state of the bit before setting to the destination register. Executing a **TAS** on a non-present word that doesn't have the sentinel value stored in it is an error and will result in a read exception.
TAC	Test and clear. Clears the presence bit of a particular word and returns the state of the bit before clearing to the destination register. If clearing the bit of a present word it writes the sentinel value into the non present word. As with **TAS** a **TAC** of a non-present word that does hold the sentinel value results in an exception.
MYAF	AF computation, given a register number computes the address of the register in the AF of the thread that executes the instruction.

Table 4.1: The new Mamba instructions

4.2.3 Initialisation, exception handling and cleaning up

Presence bits present a new hazard with regards to uninitialised memory or memory with uncertain contents. Simply loading or storing an address which has a presence bit in an uncertain state may cause unwanted side effects. If it is not present then a load may cause a thread to wait forever (as potentially nothing may ever store to it) and a store may trigger a read response to be sent when one is not needed (if the word has a value other than the sentinel value in it when not present). To avoid these issues when first using a piece of memory the presence bits must all be set before any loads or store occur to that memory. A special set presence bit operation is provided to do this. It sets 256 presence bits of 256 words aligned on a 2 kilobyte boundary at a time. The operation is triggered via a move to coprocessor operation.

Non-present registers also present a difficulty for exception handling. An exception handler should save any registers it needs to use before altering their contents so their value can be restored later. However if any of those registers is non-present the thread will wait until they become present which will take an indeterminate amount of time. To avoid this an exception handler may use three reserved registers k0, k1 and k2. These are not saved in the AF but are present in the register file (they are register numbers 26, 27 and 28, the LO, HI and ESW are stored in these AF slots but they are elsewhere when an AF occupies a context so there are three spare spaces in the register file that can be used for k0, k1 and k2). An exception program counter (EPC) is also maintained when an exception occurs which gives the instruction to jump back to when the exception handler is done. This is also not saved in the AF. Because of the existence of the EPC and reserved kn registers when an exception is triggered in a thread it is pinned into the context it occupies and will not be swapped out until the exception handler is done.

Finally there is also a hazard with killing and reusing AF memory. If an AF simply added itself to the back of some free memory list when it was done and then killed itself a race condition is introduced. Another thread could potentially grab the AF from the free memory list and reuse its memory for some other purpose before it properly kills itself. In this case the AF may still be in the register file or may be scheduled later and as described above the result of performing a load or store on such an AF is undefined. When a thread wishes to kill itself it must first ensure that all of its registers are present. It could do this by issuing a series of add instructions with the zero destination register (which would discard the result) with each register as operands in turn, but usually either all registers will be known to be present at the point the thread kills itself or those that might not be present can be confined to a small subset. Then the thread's AF can be added to a free list. A special operation (conducted via a move to coprocessor instruction) is introduced that removes the AF from the register file but doesn't actually copy any register values over nor stop the thread. The result of this is any load or store targeted at the AF that occurs after this instruction will be sent to the memory which is safe. After this instruction has occurred the thread cannot perform any further loads (any response from the load would not come back to the register file), it can use a final store to set a flag which will indicate that the AF newly added to some free list is now safe to reuse.

4.3 The Mamba system

A Mamba system consists of a number of nodes connected in a grid network (see figure 4.5). Each node comprises of a processor core, data, presence bit and instruction caches and two network routers. In this section we discuss the micro-architecture and implementation of the processor core in detail. The cache is a standard direct-mapped design so doesn't warrant much discussion. The network design and implementation was taken from elsewhere so its characteristics and its use by a Mamba node is discussed, but the actual implementation is only briefly presented.

The major parts of a Mamba node are:

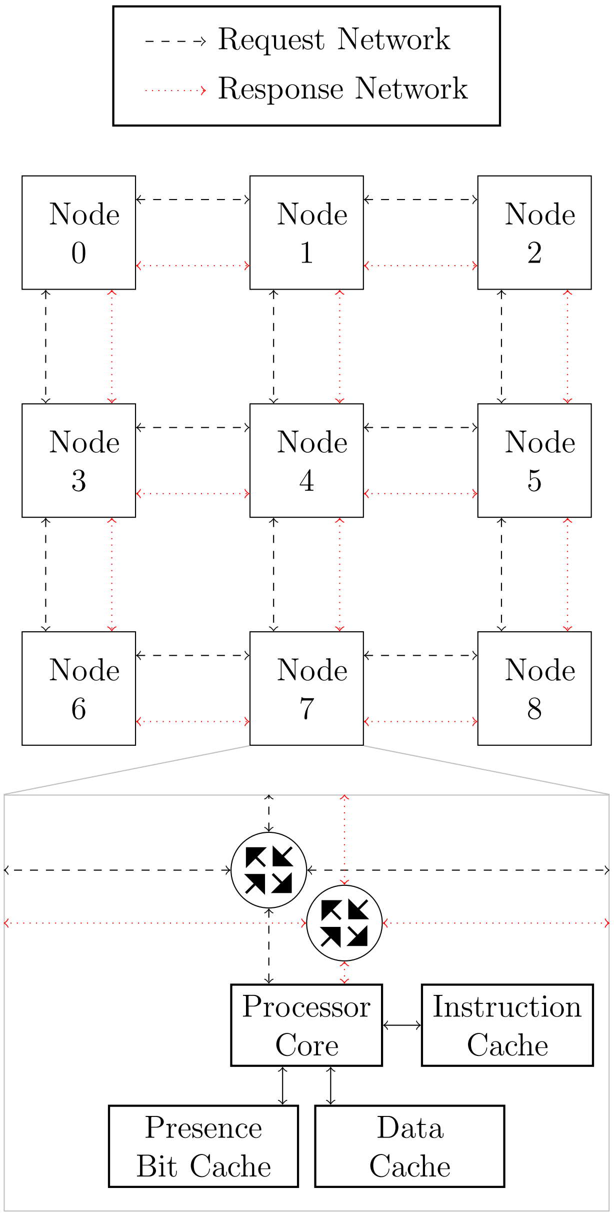


Figure 4.5: A 3x3 Mamba System

The processor core A simple in-order core implementing a MIPS64 ISA with a 4-stage pipeline. It has 8 register files so 8 threads can be run at once in a fine-grained manner.

The instruction, data and presence bit caches The caches are identical in structure. They are direct mapped, with a 32-byte line size (the same size as a DDR burst), write back with write on fetch policy. The data and instruction caches are 16KB each, the presence bit cache 4KB.

The network routers To avoid deadlock two networks are used, one for requests, the other for responses. The two routers connect the node into these two networks.

4.4 The processor core

A detailed diagram of the processor can be seen in figure 4.6. The diagram shows the data flow inside the core, which has been classified into three types

Memory Actions $\dashrightarrow$ A memory action is a packet of data that describes an action to be performed upon memory (for example 'read doubleword' would be one possible memory action) and may represent a request or a response. A response will be generated by certain requests (for example any read will generate either a response with data resulting from the read or a read exception). All memory actions have a target address. They are described in further detail below.

Internal Data $\longrightarrow$ Data flow inside the core excluding memory actions. This includes data going between pipeline stages and reads and writes to caches.

External Data $\cdots\rightarrow$ Representing a connection external to the core, the request and response networks, the DDR memory system and the Avalon IO bus are all external systems that are connected in to the core.

The core consists of several parts, brief descriptions of them are:

Scheduler, Context Queue, Ready Queue, Sequencer The scheduler chooses which of the 8 running threads to begin executing in a given cycle as well a deciding when to swap a currently running thread for a ready thread. The context queue contains contexts which represent a currently running thread. The ready queue contains AF addresses of threads that are ready to run. In order to support a number of threads bounded only by memory size the ready queue can spill over into DDR memory. The sequencer is responsible for managing a context switch when we swap a currently running thread for a ready thread.

IF,DE,EX,WB These are the four pipeline stages.

RF This is the register file, it contains 8 copies of each register and 8 program counters, one for each context.

Local Requester This is responsible for routing memory actions from several sources (sequencer/EX stage, network, RF, local memory) to the correct destination.

Local Memory This takes memory actions destined for this core's node and enacts them.

I$, D$, P$ These are the caches, the instruction, data and presence bit caches respectively.

Prof This is the profiling unit, it looks at completed instructions and updates various performance counters as appropriate.

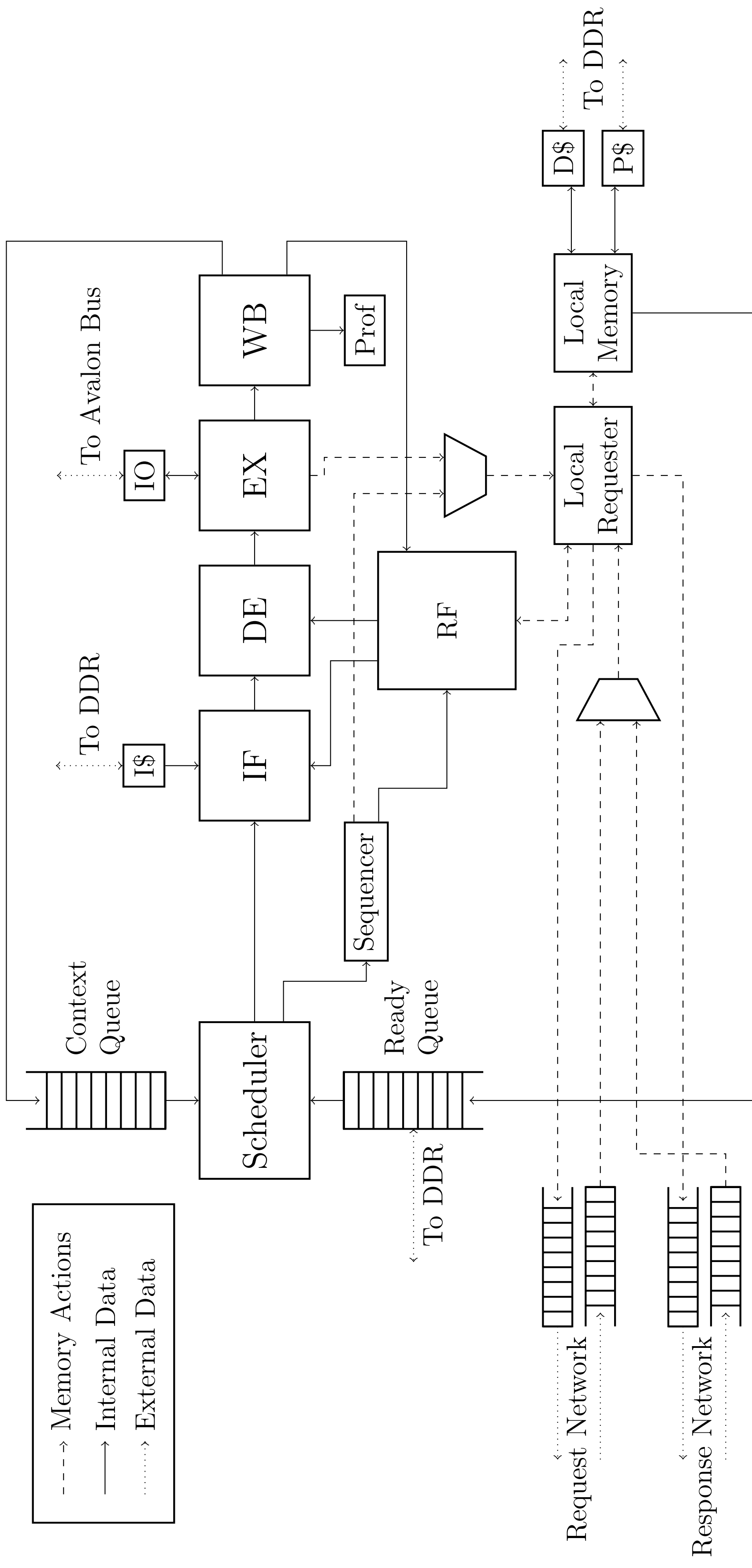


Figure 4.6: The Mamba processor core, lines and arrows illustrate the data-flow, control is omitted

4.4.1 The pipeline

The core will execute up to 8 threads at once. This is accomplished by having 8 separate register files and program counters. A context is a particular register file and program counter (PC). So a currently running thread occupies one of the 8 contexts. The scheduler is responsible for deciding which context should enter the pipeline in a given cycle. It does this in a simple round-robin order. Context numbers are enqueued into the Context Queue, the context to be run for a given cycle is simply the one at the head of this queue. Once a context enters the pipeline its number is only enqueued back on to the tail of the context queue once the instruction from that context has been through the pipeline. By doing this we ensure no data or control hazards occur because a given context will have either zero or one instructions in the pipeline at any point. Scheduling is discussed in further detail below.

If a particular instruction is unable to complete for whatever reason (A source register may not be present, or it may not able to generate a needed memory action or IO request) then the PC for the associated context is not updated, so the instruction will be tried again until it succeeds, effectively stalling the context only, not the entire pipeline.

The core is built around a 4-stage pipeline, the stages are:

IF - Instruction Fetch Given a context number from the scheduler, fetches the instruction pointed to by the current program counter (PC) of that context.

DE - Decode Decodes the instruction and fetches its source registers.

EX - Execute Executes the instruction. The execute stage includes multiplier and divider units. The multiplier is implemented with multipliers built into the FPGA so has the same latency as any other arithmetic or logic instruction (1 cycle). The divide unit has many cycles of latency (up to 64) and is not shown in detail on the core diagram. The execute unit will generate any memory action or IO access needed to execute the instruction.

WB - Write Back Writes the result of execution (or IO access) to the appropriate destination register, sets the presence bit of the destination register as needed and updates the program counter for the appropriate context.

The scheduler is not considered a pipeline stage. This is because there is no pipeline register between the scheduler and the IF stage, it provides a context number to the IF directly.

Memory access

Notably the pipeline is missing a memory access stage, this is because one is not required. When executing a load or a store the execute stage generates a memory action that performs the operation. The write memory action simply gets accepted and completed with no further interaction required with the pipeline. The read memory action has a response address where a read response is sent, so a load gives the address of the load's destination register as the response address (as any register may be referred to by its address within an AF). It is the read response that writes the result of the load back to the register file rather than the write back stage, so a memory access stage isn't needed as we don't need to wait for a load to complete to write it back. When a load is executed we clear the presence bit of the destination register, so when we attempt to use the result of the load the context will stall if the read response has not yet come back. The memory system is structured such that a load that hits in the cache will write its data into the register file via a read response before the thread that triggered that load will execute its next instruction. This is described in further detail below in the section on memory actions.

When any instruction is executed the presence bit of the destination register must also be checked. If it is not present then the context must stall until it becomes present. The reason for this is a non-present register will be awaiting a read response from somewhere (the result of a previously executed load instruction), if another instruction writes to the register before the response appears a WAW (write after write) hazard will be introduced.

Presence bit and ready queue overflow storage

The presence bit cache caches the state of presence bits however it cannot hold all of the presence bits for a node's memory so they must be stored in main memory. The memory used is the top $\frac{1}{64}$th of the node's memory. This means that area of memory cannot be written to via any running program (giving us small gaps in memory next to the borders between each node's local memory space).

In order to place no restriction on the number of threads available at a node, other than the ultimate restriction of memory size the ready queue, which holds the addresses of AFs that a ready to run can spill over into main memory. This requires an area of memory to spill in to. The next $\frac{1}{64}$th of memory below the area used to store presence bits is the spill over area. As an AF address is 256-byte aligned and only the local part of the address is required only 32-bits are required to store ready AF addresses (which gives an upper limit of 1TB memory per node, any more and more than 32-bits would be required to store the local part of each AF address). This means that $\frac{1}{64}$th of the local memory space is sufficient to store all possible AF addresses (i.e. the upper limit of the ready queue is as many AFs as we could possibly have).

Were Mamba to include a virtual memory system this could be used to hide the gaps caused by the presence bit storage and ready queue spill over area. The ready queue spill over area could also be more finely managed, the core would trap to an OS when the ready queue runs out of spill over storage and the OS could allocate it a new page to add to its available spill over area. Then the memory used by the ready queue would be of a more suitable size. The scheme detailed above is used because of its simplicity and ultimately is of not much relevance to the appraisal of the architecture.

Exception handling

If a context needs to jump into an exception handler two bits are set in the register file that indicate (i) the context is handling an exception and (ii) that the next time an instruction is fetched for that context it should be fetched from the exception handler address and not the program counter. At the same time the fetch of the first exception handler instruction occurs the PC that would have been fetched from is saved to the EPC (exception program counter) register for that context and the fetch from exception handler address bit is cleared. The handling exception bit stays set until an **ERET** occurs (this will jump to the EPC for the context resuming execution after the exception handler has finished). Any context with the exception bit set cannot be swapped out.

The register file

A diagram of the register file main storage implementation be seen in figure 4.7. It needs to support three register reads and two register writes per cycle. This is because two reads are required for up to two source registers per instruction and one write for a single destination register per instruction as well as a single read and write required to deal with any memory actions incoming to the RF.

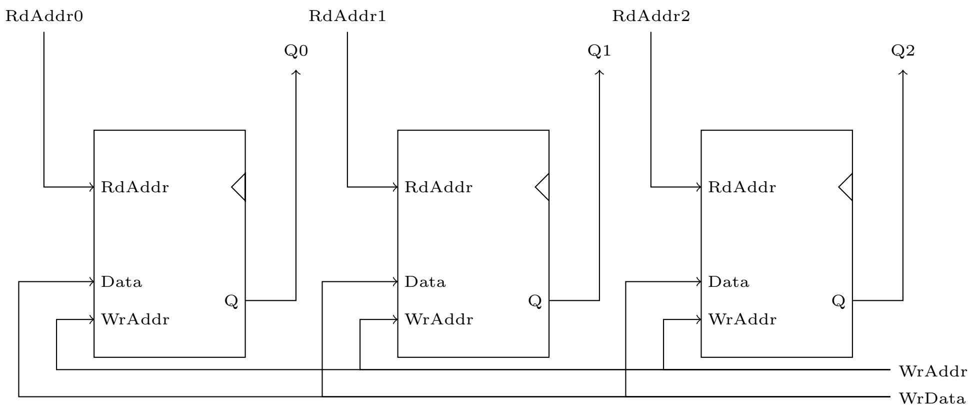
It is implemented using 8 banks (figure 4.7b) (one per context) consisting of three cells per bank (figure 4.7a). Each cell is an individual FPGA memory block. The memory blocks used support a dual-port operation that allows one asynchronous read and one synchronous write to occur at the same time, so by having three with the write ports connected together we obtain a memory with 3 asynchronous read ports and one synchronous write port. By enforcing that the two RF write ports must write to two separate contexts if both are used at the same time we can perform two simultaneous writes to two separate blocks to implement two write ports.

The first write port is used by the Write Back stage of the pipeline, the second write port is used to service any memory actions that would require a write to the register file. To ensure we never attempt to write to the same context with both write ports at the same time, the Write Back stage blocks any memory action that would target a context's registers from entering the register file if the Write Back stage will be writing to that context's registers this cycle.

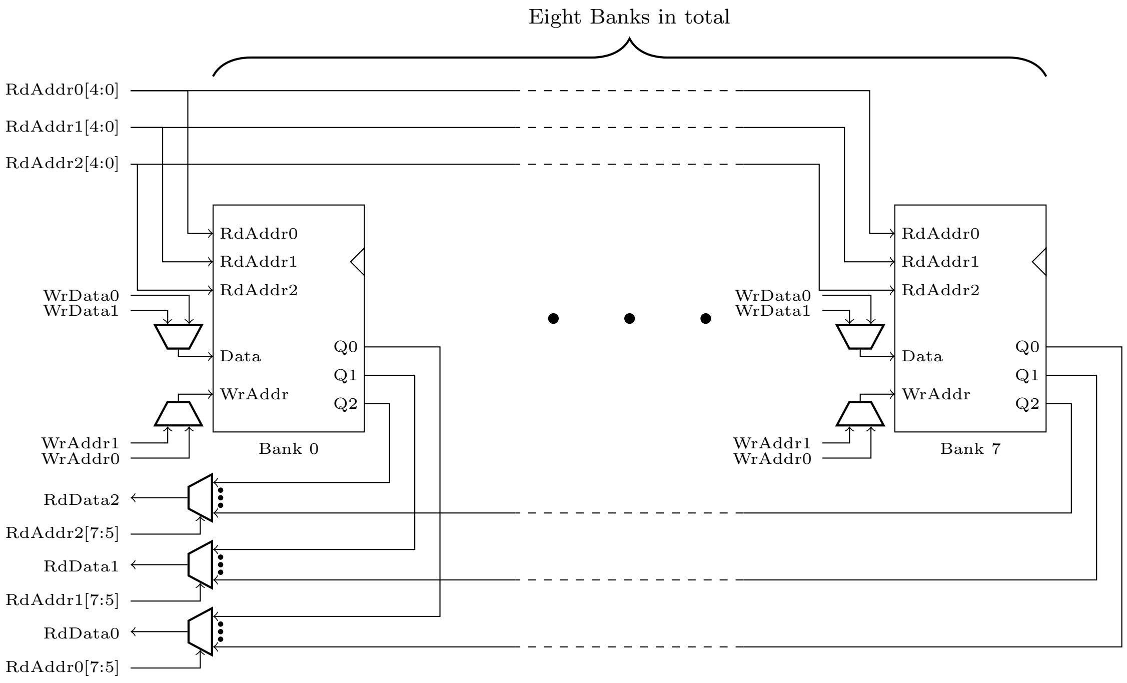
The main storage is addressed with 8-bit addresses, these consist of the register number we wish to access in the lower 5 bits concatenated with the context number in the upper 3 bits.

Along with the main set of 32 registers per context the register file also stores a program counter, exception status word (ESW) and activation frame address for each context. These are both accessed differently to the main registers so aren't part of the main storage structure shown in figure 4.7. Instead they are stored in FPGA registers (doing so for all registers would use an excessive amount of FPGA resources). Presence bits are also stored using FPGA registers, we require 32 per context, so eight 32-bit registers are employed to store them.

When a thread, represented in memory by an activation frame, occupies a context, the register file can be seen as acting as a cache for the area of memory used by the activation frame. Any memory actions targeting that activation frame must be sent to the register file so they can be performed. The local requester is responsible for determining what actions should be sent to the register file. So the register file gives it access to the



(a) Register File Main Storage bank structure



(b) Register File Main Storage constructed from eight banks

Figure 4.7: The Register File Main Storage, write enables, write addr and write data select control signals and clock connections are omitted. RdAddr$n$ is the address for the nth read port, RdData$n$ is the data. WrAddr$n$ and WrData$n$ are the address and data for the nth write port

activation frame address for each context so it can check the target of all memory actions to determine if it should be sent to the register file. The memory actions that may act upon a activation frame that is currently in the register file are restricted to avoid issues that would be caused by arbitrary reads and writes altering register contents whilst a context is running. This is discussed in further detail below in the section on memory actions.

Also of note are the HI and LO registers. These are the target of any multiply or divide instruction, holding the high and load words of a result in the case of a multiply or the quotient and remainder in the case of a divide. These are not present in the register file main storage either, this is because (i) doing so would give the banks in the main storage a non power of two size, (ii) after a multiply or divide we must write to both at the same time, and it is not allowable to use both write ports of the main storage to write to the same bank and (iii) HI and LO cannot be generally used as a source register nor can they be the target of any memory action, so the 3 separate read ports and extra write port are not necessary. Again FPGA registers are used to store HI and LO for each context.

4.4.2 Memory actions

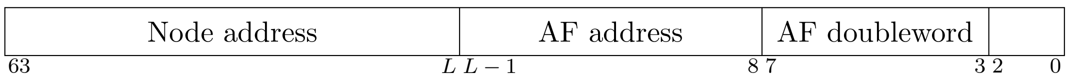


Figure 4.8: The parts of a memory action's target address, L refers to the number of bits in the local part of the address so 2^L is the size of a node's local memory area

A memory action is targeted at a particular address (the target address) which refers to a 64-bit word memory (the target location) and gives an operation to perform related to that location. A target address can be split up into three parts, node address, AF address and AF doubleword as shown in figure 4.8. The bottom 3 bits of the address are ignored when determining where to send the action, they are only utilised for read and writes of non double-word size to determine what part of the 64-bit word needs to be operated upon.

A memory action may be generated by one of four things, (i) the execute stage of the pipeline, (ii) the sequencer, for transferring registers during context switches and (iii) the local memory and (iv) the register file, in response to a another memory action. Any generated memory action goes to the local requester, which decides where to send the memory action. A memory action's destination will depend upon whether or not its target address is local to the node and whether or not it refers to an AF that is currently occupying a context in the register file. The request and response networks are also a source of actions, but neither network generates actions of its own accord.

A memory action falls in to one of five categories, which are described below, with pseudo-code describing their semantics in listings 4.1, 4.2, 4.3, 4.4, 4.5. The pseudo-code describes how a particular action is processed. In the pseudo-code, data refers to the data read from the target address, present is true if the target location is present, type refers to the type of the memory action, AF address and doubleword refers to the AF address and doubleword parts of the target address (figure 4.8), AF presence is a 32-bit word containing all of the presence bits in the AF referred to by the target address. Other variables are taken from the memory action itself. In the descriptions below a tuple is given of the data in each action.

A memory action may be processed by either the register file or the local memory depending upon whether or not its target is in an area of memory that is part of an AF currently occupying a context. If an action is processed by the register file the writeMem, readMem and setPresence functions in the pseudo-code are acting directly on the relevant registers, not memory. Otherwise they are operating upon memory via the local node's data and presence bit caches.

Reads Action data consists of (target address, response address, size, trigger exception). A request to read the data at the target address. It will result in an immediate read response sent to the response address if the location is present and either no immediate response or a read exception if the data is non-present (depending on if a forwarding address is already set for the location or not). A read size that is not a double-word will result in a read exception sent to the response address if the location is not present (presence bit semantics only work on 64-bit words, not parts of that word).

A read may be marked as a vacating read (trigged with the **VL** instruction) in which case if the target location is present the presence bit will be unset as well as sending a read response. In the case of a non-present location a vacating read always results in a read exception.

The trigger exception flag is set for a read triggered by an **LD** and unset for a read triggered by an **LDNR** (its value is always 0 for a vacating load as they never trigger the exception handler). The trigger exception flag is copied into any read exception generated by the read, which causes different behaviour in the AF that receives the exception (see 'Responses' for details).

Details are given in pseudo-code in listing 4.1.

Writes Action data consists of (target address, write data, size). A request to write data to the target address. If the target location is present it is updated with the write data and nothing further happens. If not present it may cause a read response to be generated if a target has a valid forwarding address. A write will set the presence bit of the target location. If the write is of non-doubleword size then the write will update the target location and set the presence bit but if there is a forwarding address then it will send a read exception rather than a read response (as writing a non-doubleword to a non-present location should not be done).

A write may be marked as a write to an activation frame (trigged with the **SDA** instruction), in which case all the presence bits in the activation frame that contains the target location will be checked when the write occurs. If they are all set then the activation frame is ready to run and it gets placed on the back of the ready queue.

Details are given in pseudo-code in listing 4.2.

Responses There are two types, a read response with data (target address, read data) and a read exception with data (target address, trigger exception).

A read response is much like a write to an activation frame in that it will update the target location with the read data, set the presence bit and check to see if every presence bit in the activation frame that contains the target location, adding that activation frame to the ready queue if they are all set. However it differs in two ways (i) it doesn't check for a forwarding address so it will never generate a further read response (this could lead to a deadlock in the network) and (ii) if the presence bit is already set it does nothing. However as a read response will only ever be sent due to a read request its target location will have had its presence bit unset when the read request was generated so the situation of a read response being targeted at a present location should not occur.

A read exception will set the presence bit of the target location and then set the appropriate bit of the corresponding ESW. If the memory action has been sent to the register file this will be a separate FPGA register within the register file, if it has been sent to local memory, the exception status word is in the 28th 64-bit word of the activation frame and can be written directly. As with a read response after setting the presence bit it will check all presence bits in the activation frame and add it to the ready queue if all are set. Read exceptions targeted at present locations are ignored.

A read exception should also cause the thread that owns the location it is targeted at to jump to the exception handler, except in the case where the read exception is the result of a read request action triggered by an **LDNR** or **VL** instruction. The trigger exception flag indicates whether or not a read exception should cause a jump to the exception handler or simply set the appropriate bit in the ESW. When an **LDNR** instruction is executed the presence bit of the destination register is cleared and the read request generated exactly as with an **LD** instruction but with the trigger exception flag unset in the read request. The trigger exception flag of a read exception matches the trigger exception flag of the read request that triggered it. If a read exception is targeted at a location not currently in the register file then the exception handler may have to be triggered when the AF gets swapped in, this is indicated by setting the 32nd bit of the ESW.

Details are given in pseudo-code in listing 4.3.

Presence bit operations The **TAS** and **TAC** instructions will generate a memory action with data (target address, response address) that will set or clear the target location's presence bit (depending on whether **TAS** or **TAC** is used respectively) and then return a read response with the value of the presence bit before it was changed. If we **TAS** a non-present location that doesn't have the sentinel value in then a read exception is sent to the response address instead leaving the presence bit and location untouched. The reason for this is if something other than the sentinel value is there it may be a forwarding address or it may just have been uninitialised memory, in the former case setting the presence bit without sending a read response to the forwarding address would cause a thread to wait forever for a response that never comes. In the latter case if we choose to send a read response to the potential forwarding address if it turned out to just be uninitialised memory a read response will be sent somewhere that wasn't expecting one potentially causing issues. Performing a **TAC** on a non-present location that doesn't have the sentinel value in also causes a read exception, for similar reasons. If the **TAC** reset the word to the sentinel value the thread waiting for that word to become present would wait forever. Code **TAC**ing a location intends to have a non-present location with no forwarding address, so if this doesn't actually happen an exception is appropriate.

There is also a set line presence memory action that is used to set an entire presence bit cache line's worth of presence bits (256 presence bits, corresponding to 256 64-bit words or 2KB of memory), this is used for the rapid initialisation of newly allocated memory as the use of memory with uncertain presence bits may have unintended side effects (such as spurious read responses being sent).

Details are given in pseudo-code in listing 4.4.

Register transfer operations When the scheduler chooses to swap a running thread out of a context and swap a new one in we need to copy the thread's register state, represented by its activation frame, into memory and copy the new thread's activation frame into the register file. Two separate actions are used, register transfer to memory and register transfer to register file both with data (target address, register data, presence bit). Register transfer to memory actions are generated by the register file during a context switch and register transfer to register file actions are generated by the sequencer.

A register transfer to memory action simply writes the register data into the target location and sets the presence bit to the given value, if the target location corresponds to the final word in an activation frame all the presence bits of the activation frame are checked, if they are all set the activation frame is ready and it is placed at the back of the ready queue.

A register transfer to register file action acts differently depending upon whether it is processed in local memory or the register file. The sequencer generates the action initially and it is sent to the local memory. When local memory processes the action it reads the data and presence bit in the target location and generates a new register transfer to register file action with the read data and presence bit. This then goes to the register file, the register file updates the register corresponding to the target location with the given register data and presence bits. This is discussed in further detail in the scheduling section below.

Details are given in pseudo-code in listing 4.5.

The local requester

The local requester is responsible for routing memory actions around the core. Any memory action that has either been generated in the core, or is coming in to the core from the network goes to the local requester which routes it to the correct location. Conceptually this is a simple process; first the node address part of any memory action's target address (figure 4.8) is compared to the local node's address, if they match the action is targeted at this node, otherwise it is targeted at a remote node and sent out to the network. If it is targeted at the local node a comparison against the AF address part of the target address is done against the 8 AF addresses of the threads currently occupying the register file. If the AF address matches any AF currently in the register file the memory action is sent there (with the address rewritten to refer to a context number instead so the register file need not repeat the AF matching procedure), otherwise it is sent to the local memory.

In practise the routing is a delicate procedure, we need to ensure low latency whilst also avoiding deadlock.

There are four sources of memory actions connected in to the local requester:

```
if(size == doubleword) {
        if(present) {
                generateAction(Read Response, (response address, data))
                if(type == Vacating Read) {
                        clearPresence(target address)
                }
        } else {
                //1 is the sentinel value, an invalid forwarding address.  If it's
                    present in the target location then we can write in our respose
                    address as the forwarding address, however this is not done on a
                    vacating read
                if(data == 1 && type != Vacating Read) {
                        writeMem(target address, response address)
                } else {
                        //For a vacating read exception flag is never set as it never
                            triggers the exception handler
                        if(type == Vacating Read)
                                generateAction(Read Exception, (response address, 0))
                        else
                                generateAction(Read Exception, (response address,
                                    exception flag))
                }
        }
} else {
        if(present) {
                //Select from the 64-bit data we have read the part we need given the
                    bottom 3 bits of address and read size
                data = selectFromDoubleWord(data, target address[2:0], size)
                generateAction(Read Response, (response address, data))
        } else {
                //Exception flag always set if non doubleword read as it should always
                    trigger the exception handler
                generateAction(Read Exception, (response address, 1))
        }
}
```

Listing 4.1: Read memory action pseudo-code

The execute stage and sequencer (they are multiplexed into one source), referred to as the CPU source below

The register file

The local memory

The network (we have two networks, request and response these are multiplexed into one source)

The are three destinations the local request may send a memory action:

The register file

The local memory

The network (With requests and responses sent on two different networks)

Each destination will only accept a single memory action per cycle, so arbitration is required to determine which memory action goes to a destination if several need to be routed to the same one. This is done with the following static ordering:

```
if(present) {
        writeMem(target address, write data, size)
} else if(size == doubleword) {
        //If we don't have the sentinel value we have a forwarding address, so generate
            a read response to that address
        if(data != 1) {
                generateAction(Read Response, (data, write data))
        }

        writeMem(target address, write data, size)
        setPresence(target address)

        if(type == Write to Activation Frame) {
                //If all bits in the AF are now set write AF address to ready queue
                if(AF Presence == 0xFFFF_FFFF)
                        addToReadyQueue(AF address)
        }
} else {
        //Present bit semantics not designed to work with non doubleword size
            operations, so a non doubleword write to a non present location generates an
            exception if we have a forwarding address (as it was expecting a doubleword)
        if(data != 1) {
                generateAction(Read Exception, (data, 1))
        }

        writeMem(target address, write data, size)
        setPresence(target address)
}
```

Listing 4.2: Write memory action pseudo-code

1. Register File
2. Local Memory
3. CPU/Network

As ultimately all memory actions are generated due to the actions of a processor core, the register file and local memory are unable to prevent the CPU and network from accessing a destination forever so their higher priority does not cause a starvation issue.

To minimise latency routing decisions are made with purely combinational logic. All memory action destinations have a signal that goes high to indicate an action sent to them has been accepted (if isn't accepted the action needs to be held by the source until it can be, this will block the source from sending any other actions as well). However for the local memory and register file their action accepted signal depends in part on whether the action they may wish to send this cycle has been accepted. As the register file and local memory may wish to send actions to each other, or the local memory may wish to send an action to itself this introduces a loop in the logic if only the action accepted signals are used to determine whether an action is accepted (e.g., local memory may try to send an action that is routed to itself, in which case it will only accept the incoming action if its outgoing action is accepted but its outgoing action is only accepted if its incoming action is accepted).

To break this cycle two extra signals are introduced, an 'all actions blocked' signal and an 'accepts any actions' signal. If 'all actions blocked' is raised by a destination then any action sent to it will not be accepted, if 'accepts any actions' is raised by a destination

```
if(type == Read Response) {
        //A read response to a present location is ignored
        if(!present) {
                writeMem(target address, read data)
                setPresence(target address)
                //If all bits in the AF are now set write AF address to ready queue
                if(AF Presence == 0xFFFF_FFFF)
                        addToReadyQueue(AF address)
        }
} else if(type == Read Exception) {
        //A read exception to a present location is ignored
        if(!present) {
                setPresence(target address)
                //If the target is in the register file we have a exception status word
                    register we can write directly
                if(RF) {
                        //Bits 7 - 3 of the target address are the number of the
                            register within the AF that is the target of the read
                            exception
                        setExceptionStatusWordBit(AF doubleword)
                        //If exception flag is set we trigger the exception handler
                        if(exception flag) {
                                triggerReadException()
                        }
                }
                //Otherwise in local memory we write to the exception status word which
                    is stored in the AF in the 28th word
                else {
                        //Exception status word address found by concatenating AF
                            address with 0x1C = 28 in decimal and setting bottom 3 bits
                            to 0
                        exceptionStatusWordAddress = {AF address, 0x1C, 0}
                        exceptionStatusWord = readMem(exceptionStatusWordAddress)
                        setBit(exceptionStatusWord, AF doubleword)
                        if(trigger exception) {
                                //32nd bit of ESW indicates that AF needs to jump to
                                    exception handler when its switched in
                                setBit(exceptionStatusWord, 32)
                        }

                        writeMem(exceptionStatusWordAddress, exceptionStatusWord)

                        if(AF Presence = 0xFFFF_FFFF)
                                addToReadyQueue(AF address)
                }
        }
}
```

Listing 4.3: Response memory actions pseudo-code

than any action sent it to will be accepted. If neither is raised then the design of the local memory and register file guarantees that any action sent to them will be accepted if and only if their own outgoing action is accepted, or if they have no outgoing action.

The routing logic thus proceeds as follows:

1. Examine the target addresses of the memory actions from all sources and determine which destination they should be routed to.
2. If more than one source wishes to send an action to the same destination, apply arbitration as specified above.
3. After arbitration each destination will have zero or one actions to be sent it, we then determine what actions will be accepted.

```
if(type == TAS || type == TAC) {
        //If we're trying to set and location is not present and it doesn't contain the
            sentinel value then this is an invalid TAS so send a read exception
        //If we're trying to clear and location is not present and it doesn't contain
            the sentinel value this is also invalid.
        if(data != 1 && !present) {
                generateAction(Read Exception, (response address, 1))
        } else {
                generateAction(Read Response, (response address, present))
                if(type == TAS) {
                        setPresence(target address)
                } else {
                        clearPresense(target address)
                        //Write in sentinel when clearing presence (otherwise it may
                            contain an arbitrary forwarding address)
                        writeMem(target address, 1)
                }
        }
} else if(type == Set line presence) {
        //This sets the presence bits of 2KB worth of memory, so we discard the bottom
            12 bits of the target address
        setLinePresence(target address[63:12])
}
```

Listing 4.4: Presence bit memory actions pseudo-code

```
if(type == Register Transfer to Memory) {
        writeMem(target address, register data)
        setPresenceTo(target address, presence bit)
} else if(type == Read Transfer to RF) {
        if(Memory) {
                //If this action is being processed by local memory we generate another
                    Read Transfer to RF action, with the data and presence bit found at
                    the target address. This will make its way to the register file
                    where the data and presence read here will be written into the
                    register file
                generateAction(Read Transfer to RF, (target address, data, present))
        } else {
                //Read Transfer to RF being processed by the register file, so it
                    contains new register data and presence read from memory
                context = contextNumberFromAF(AF address)
                writeRegister(context, AF word, register data)
                setRegisterPresenceTo(context, AF word, presence bit)
        }
}
```

Listing 4.5: Register transfer memory actions pseudo-code

If a destination has raised ‘all actions blocked’ the action won’t be accepted.

If a destination has raised ‘accepts any actions’ the action will be accepted.

For the network destination, then we use the network ‘action accepted’ signal as there is no circularity (Whether or not the network can accept depends only upon how many elements are in the outgoing FIFO).

For the local memory and register file destinations it is more complex. We need to examine any outgoing action they have as well to determine whether an incoming action will be accepted. If the local memory or register file is sending to the network then they can accept an incoming request if the network can accept their outgoing action, provided they haven’t raised ‘all actions blocked’. If the local memory is sending to itself it will always be accepted, provided

```
if(memory all actions blocked) {
        //Memory blocked all actions, cannot accept action
        return false
} else if(memory accepts any actions) {
        //Memory accepts all actions, can accept action
        return true
} else if(no memory outgoing action) {
        //If memory is not sending any action it will accept an action
        return true
}
//Memory is sending an action, can only accept an incoming action if outgoing action is
    accepted
else if(memory outgoing destination == network) { //Memory outgoing action will go to
    network
        return network action accepted //So can accept incoming action if network
            accepts outgoing action
} else if(memory outgoing destination == memory) { //Memory sending to itself
        return true //Can always accept in this case
}
//Memory is sending an action to RF, so can only accept an incoming action if RF will
    accept our outgoing action
else if(memory outgoing destination == RF) {
        if(rf all actions blocked) //RF blocking actions
                return false //Can't send memory outgoing, so memory incoming is not
                    accepted
        else if(rf accept any actions) //RF accepting actions
                return true //Memory outgoing can be sent so memory incoming can be
                    accepted
        else if(rf outgoing destination == memory)
                return true //If RF sending to memory and memory sending to RF can
                    accept incoming
        else if(rf outgoing destination == network) //If RF sending to network
                return network action accepted //If RF can send to network then memory
                    can send its outgoing so it can accept its incoming
        else if(no rf outgoing action) //If RF sending nothing
                return true //Then memory outgoing can be sent so memory incoming can be
                    accepted
}
```

Listing 4.6: Local Requester accept logic for local memory destination, logic is similar for register file

'all actions blocked' isn't raised. If local memory is sending to the register file then we need to check if the register file has an outgoing request and determine whether or not that will be accepted. The full detail of this is given is pseudocode 4.6.

An action traverses the local requester with zero cycles of latency. This is important as a load instruction is implemented by the execute stage generating a read memory action with the response address set as the address of the load's destination register. If the execute stage generates the read request memory action on cycle n then provided the cache isn't busy and the action isn't blocked by one of a higher priority (i.e. one from the register file or the local memory) then the cache read will start cycle n. If we have a cache hit then the data will be returned from the cache on cycle $n + 1$ (when the instruction that triggered the load is in the write back stage), and the local memory will generate a read response action with the returned data. This be sent to the register file on cycle $n + 2$, with the data written to the register file on the same cycle (due to the zero cycle latency of the local requester), as this point the context that triggered the load will have just been written to the context FIFO. This ensures that the result of the load is written to the register file before it is needed (if the instruction following the load uses the result immediately) if we have a cache hit avoiding a needless stall.

The write back stage has the ability to block memory actions targeted at a particular context from entering the register file. This is used when writing a destination register in write back, we block memory actions targeted at the context the destination register is in to avoid two writes to the same context occurring the same cycle as the register file doesn't support this.

During a context switch the contents of the AF being swapped into memory and the AF being swapped into the register file could be in either place so the local requester may route requests for them to the wrong place. To avoid this all actions from the CPU and network are blocked during a context switch.

The local requester also has a couple of hard routing rules that ignore the target of a memory action. A register transfer to register file memory action coming from the CPU always gets routed to the memory and a register transfer to memory memory action coming from the RF always gets routed to the memory. This is to enable the context switch process, further details are given below in the Scheduling section.

4.4.3 Scheduling

The scheduler is responsible for two things, providing the next context to have its instruction fetched to the pipeline and deciding when to perform a context switch, where all the registers of an AF currently in the register file are copied out to memory and a new AF coped in. The sequencer is responsible for orchestrating the actual switch, the scheduler is just responsible for choosing when this happens which context should be switched and what AF will be copied into the register file.

The scheduler is pre-emptive with round robin used to choose the next AF. There are eight contexts and each cycle the scheduler must provide a context number to the pipeline which will be the next context that has its instruction fetched and executed. It takes this context number from the head of the context FIFO. When a context has finished the write back stage its context number is written to the tail of this FIFO, at the same time the quantum counter for that context is decremented.

When the quantum counter reaches 0, the deschedule signal for that context is asserted. The deschedule signals are fed into a priority encoder which chooses which context to deschedule if several have their deschedule signal's asserted. The context with the lowest number will be descheduled before a context with a higher number.

If a context is handling an exception it must not be descheduled, this is because during the execution of the exception handler it may use the special k0, k1 and k2 registers that are not saved into the activation frame so there is no way to resume execution of the exception handler if an AF is swapped out of a context whilst it is handling an exception. A 'handling exception' bit is set in the register file for a context for the entire time the context is handling an exception, if that bit is set the scheduler will not deschedule the context.

When a context to be descheduled has been chosen we take the AF at the head of the ready queue and notify the sequencer of the context number that is being switched and the AF to be switched in. The actual switch is started when the write back stage attempts to add the number of the context to be descheduled to the back of the context FIFO, this is suppressed and the switch started, when the switch is finished the context number is added back to the context FIFO again. This is done to ensure the switch doesn't start until the context is not in the pipeline and that the context isn't executed whilst it is being switched.

Sequencer

The sequencer is responsible for orchestrating the context switch. It is supplied with a context number and the address of the new AF to switch in. First it notifies the register file that a switch is occurring and tells it to start generating register transfer to memory memory actions which transfer the contents of the context's registers to the appropriate AF in memory. Due to the hard routing rules in the local requester these are sent to the local memory despite the fact that they target an AF still in the register file. Once all of the context's registers have been transfered the AF address of the context is updated to the new AF. The sequencer itself then starts generating register transfer to register file actions, these are given a target address of each of the new AF's registers in turn. Again due to the hard routing rules in the local requester these are sent to the local memory, despite the fact they target the new AF which now has its address written into one context's AF address register. In local memory the transfer action causes the register contents and presence bit to be read from memory and generate another register transfer to register file memory action with the same target address using the data just read. This gets routed to the register file due to the standard routing rules. Once this has been done for all registers the context swap is complete and the number of the context we were swapping can get written back to the context FIFO so it can be executed again.

During the context swap process all memory actions from the execute stage and from the network are blocked, this is for two reasons

> We do not know where to send a memory action being targeted at an AF being swapped. Furthermore a memory action that writes may trigger a read response that targets an AF being swapped so simply blocking any action from the execute stage or network that targets one of the AFs being swapped is not sufficient as we may have block further actions generated in the local memory and blocking local memory would prevent the context swap from finishing.
>
> It allows the context swap to finish in as short a time as possible.

Also before the context swap process is started we must wait for any pending memory actions in the local memory to be completed in case one of them targets one of the AFs being transfered (or generates a memory action targeting one of the AFs being transfered)

4.4.4 The caches

All of the caches are direct mapped, they are write back caches and implement a fetch on write policy. Lines are 256-bits long, the same size as a burst read or write from the DDR memory.

They are unremarkable apart from two features:

Whole Line Set When a chunk of memory is first used its presence bits are in an unknown state. Operating upon memory with presence bits in an unknown state is hazardous as writes may trigger unwanted read responses (the data in the memory chunk is also undefined so something that is non present may have what appears to be a forwarding address within it), and reads may never generate a response (though reading memory you have yet to write to could also be considered hazardous). To resolve this we need a mechanism to set presence bits, preferably in large batches to avoid spending too long on memory allocation. A single line in the presence bit cache holds 256 presence bits, the whole line set mechanism sets all of the bits within a cache line to zero or one. If the whole line set is targeted at an address not in the cache then the new line is written directly to main memory without fetch (a violation of the fetch on write policy, however if using a whole line set it is likely the line won't be immediately accessed as it's probably part of an initial memory allocation, so it would be counter productive).

Immediate Write When performing a read memory action if the location is non present and there is no current forwarding address the read actions response address must be written to the location. A write to the cache requires a line to be fetched from the cache memory (to check the tag) so starting a fresh write would cause a repeat of the last cache memory read. Immediate write allows the line read in the cache read from the previous cycle to be altered and written back to cache memory. This mechanism is also used for all write memory actions as they may trigger read responses, so need to read the line before updating it anyway.

4.4.5 The network

The network design and implementation was not the work of the author. It is briefly presented here for completeness. The network was built by Arnab Banerjee based on his work [10].

Two separate networks are employed, a request and a response network, they are identical in design. This is required to avoid protocol deadlock. Each network has a 2-dimensional mesh topology. At each node there is a router, this router has 5 inputs and 5 outputs. The first 4 inputs and outputs are north, east, south and west, they connect the router to the other routers in the topology. The fifth input and output is tile, this connects to the Mamba node. Data in the network is passed around as flits, several flits make up a packet which carries a single memory action. The header of each flit has an output port field, this is the output port a router must send an incoming flit to. The router also calculates the output port the flit must be sent to for the next router. Dimension ordered routing is used to do this. A flit first gets sent along the east-west direction of the mesh until it reaches the column that contains its destination, then it traverses the north-south direction until it reaches its destination and which point it gets sent to the tile port of a router.

When a memory action packet is generated at a node the x (east-west) and y (north-south) displacements required to get to the target node are calculated. These are placed

$$\begin{tikzcd}
\text{CTS} \arrow[r, bend left, "\text{nearly full \& flit sent}"] & \text{DNS} \arrow[l, bend left, "\text{nearly full}"]
\end{tikzcd}$$

Figure 4.9: On/Off flow control state machine, CTS is the clear to send state, DNS is the do not send state

in the header of each and a reduced by 1 towards 0 when the flit makes a step in the corresponding direction (x displacement reduced when going east-west, y displacement reduced when going north-south). A router looks at the remaining displacement required and determines the appropriate output port on the next router (east or west if we still have x displacement left, north or south if we have y displacement left and tile if no displacement is left) and writes this in to the flit header.

On/Off flow control is used between the routers. Between two routers connected by a link a there is a single signal, nearly full. This is asserted when the input buffer of the downstream router reaches a certain number of empty spaces n. It is only asserted when the number of empty spaces is precisely n, not when it is below n. The upstream router requires a small state machine (see figure 4.9), this starts off in the clear to send (CTS) state, when the nearly full signal is asserted and the upstream router sends a flit on the link it transitions to the do not send (DNS) state, as the flit being sent will push the incoming buffer of the downstream router over the threshold. When in this state the upstream router must send no more traffic on that link. It transitions backs to the CTS state when the nearly full signal is asserted (which will happen when the downstream router has precisely n spaces again, so something has been removed from the buffer) allowing the upstream router to begin sending on that link again.

4.4.6 Profiling

The core includes a profiling unit, which holds performance counters that can be used to measure a variety of metrics. All counters are per-core rather than per-context or per-thread. The available performance counters are:

Cache Counters There are operation counters and miss counters for each cache with separate operation and miss counters for reads and writes. Operation counters are incremented every time a read or a write is requested from the cache and the corresponding miss counter is incremented if that operations leads to a cache miss

Stall Counters There are two reasons a thread may stall (not update its program counter to the next instruction at the write back stage so the current instruction gets repeated), there is a stall counter for each of them which gets incremented every time that kind of stall occurs

No Presence Stall A stall that occurs when a context cannot progress because a needed register is not present

Memory Request Blocked Stall A stall that occurs when the execute stage is unable to generate a memory request needed to complete the instruction because there is no space in the outgoing action queue.

PC Region Counters PC regions can be defined by software, any time a context tries to execute an instruction within one of these regions the corresponding counter is incremented regardless of whether or not that instruction causes a stall or not. It does this rather than only counting successful executions as that would not give an accurate picture of how long a core spends executing a particular region.

Step Counter Every time something traverses the write back stage the step counter is incremented, this is to give a count that can be used with the other counters (e.g., to compute percentage of time spent in a region using the pc region counters).

The profiling unit receives input from the write back stage. Everything that goes through the write back stage generates a profile input which gives the PC that the instruction being executed came from, whether or not it stalled and if it did stall for what reason. This passes through a small pipeline which first compares the given PC against the PC regions and then updates the counters appropriately. The cache counters are updated directly by the cache when an operation or miss occurs.

Software has the ability to pause profiling so no counters and get updates and the ability to reset all counters at once in one atomic instruction.

4.5 FPGA Implementation

The actual system implementation utilises Altera Stratix IV EP4SGX 230 FPGAs on a Terasic DE4 board. The DE4 board consists of a single FPGA, a multitude of peripherals, two DDR2 memory channels and high speed serial interconnect, full specifications can be seen in table 4.2. There are 4 Mamba nodes per FPGA, so each DDR2 channel is shared between two nodes. An 8-core system is constructed by continuing the network links over high speed serial connections to another board. The majority of the system was written in the Bluespec System Verilog [78] HDL, with small amounts of Verilog utilised to interface with Altera supplied IP (such as the DDR memory controller and high speed serial interface) and FPGA specific features (the register file in the processor core is constructed of a specific arrangement of in-built FPGA memory blocks).

Table 4.2: DE4 Board Specifications

1 x Stratix IV GX EP4SGX230
2 x DDR2 Memory Channels, 4GB maximum capacity per channel
4 x Gigabit Ethernet Ports
2 x SATA Host Ports, 2 x SATA Device Ports

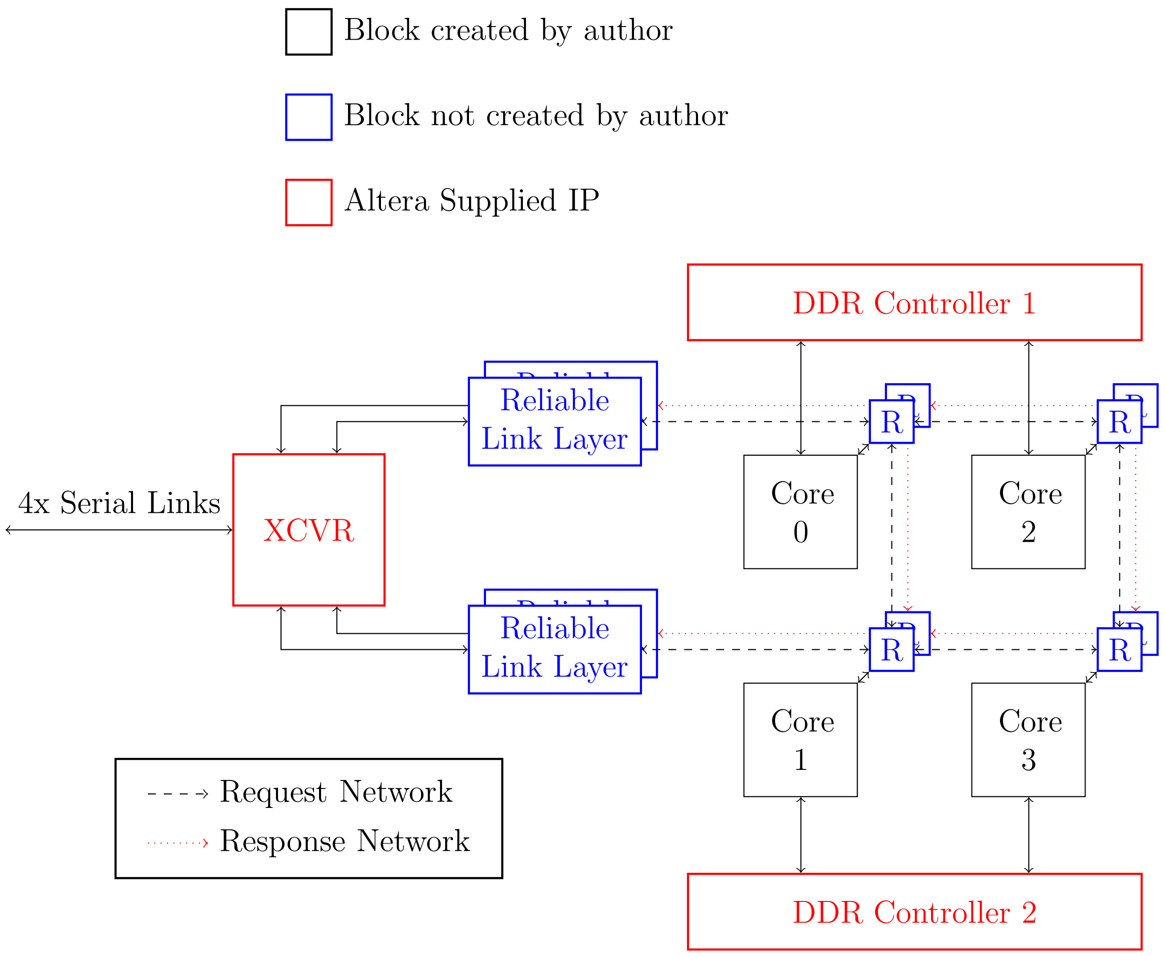


Figure 4.10: The FPGA implementation of Mamba. R signifies a network router, XCVR is the Altera component responsible for driving the serial links. There are also 4 serial controllers that communicate via JTAG over USB utilised to output messages to a host PC and a block that can read and write DDR memory controlled by the host PC which is used for programming and results gathering.

Figure 4.10 illustrates what is implemented on a single FPGA. It is a mixture of blocks written by the author, blocks written by others (Routers and network by Arnab Banerjee, reliable link layer by Simon Moore) and Altera supplied IP. The designs used by the two boards are effectively mirror images of each other so the networks match up. The SATA host and device ports provide a total of 4 serial channels in each direction. This allows each to be dedicated to a single network link without requiring any multiplexing. The reliable link layer is responsible for taking network flits and transmitting over a serial link, it implements a reliable link protocol that ensures all network flits are fully transmitted (otherwise the Mamba would have to deal with the possibility of packet loss in the network). The XCVR is the Altera supplied IP that actually drives the serial links. Altera's Avalon bus is used to connect cores to DDR controllers and the reliable link layer to the XCVR. Also present are four serial controllers that communicate via JTAG over USB, and a block capable of reading and writing DDR memory, both are supplied by Altera. The serial controllers are used for low-speed communication with the host PC so messages from each core can be displayed in a terminal. The block that reads and writes memory is used to program the system and gather results from the host PC. Neither of these are pictured in figure 4.10.

4.6 Summary

This chapter presented the architecture of Mamba. A system designed to encourage the creation of programs that will scale over a large number of cores without the need to tailor a program to a specific number of cores. This is accomplished by provided a simple, lightweight synchronisation mechanism in the form of presence bits combined with a hardware scheduler.

Chapter 5

Software techniques

Mamba's programming model gives rise to new software techniques, these are discussed in this chapter. Many are adaptations of existing algorithms and data structures altered to use presence bits.

5.1 Primitives

5.1.1 Basic spin lock

An obvious use of a presence bit is as a simple lock. A particular word could start off present, representing the unlocked state. **TAC** can be used to clear the word. **TAC** returns the state of the presence bit before it clears it and that can be checked. If it was present before the clear the lock has been successfully acquired if it was not present before the clear something else has the lock, so either the **TAC** is retried until lock acquisition is successful or the thread does some other work. This gives a basic spin-lock, see listing 5.1. However it doesn't present any advantage over a spin-lock constructed using a compare and swap instruction.

5.1.2 Acquiring locations

The presence bit of a word could be seen as a lock specifically for the contents of that word. If the word is present then the lock has not been acquired so the word may be read, but not modified. If the word is not present then some thread has acquired the lock and may be modifying the data in the word. This can be implemented with the vacating load, the **VL** instruction, which clears the presence bit of a word if the load succeeds (that is the word is present when the request is processed). If the **VL** does not succeed then an exception response is sent back. So when accessing a word in such a way first an **VL** is performed, then the exception status word (ESW) is checked to see if an exception response was received. If not the **VL**'s destination register has the data from the word and the presence bit of the word is unset. If an exception was received then the **VL** can be retried. This process is known as acquiring a location. When a thread has acquired a

```
//Acquire the basic spin lock, lock, it is a pointer to a 64-bit word
void acquire_basic_spinlock(lock_t* lock) {
        uint64_t oldPresence;
        do {
                //Test and clear the lock word
                oldPresence = TAC(lock);
        //If the old presence bit was cleared, then the lock was already acquired so we
            need to try again.  If it was set then the lock was not acquired and we have
            just cleared the lock word's presence bit, so the lock has just been
            acquired.
        } while(!oldPresence);
}

//Release the basic spin lock, lock
void release_basic_spinlock(lock_t* lock) {
        //Releasing the lock just requires setting the lock word's presence bit.
        *lock = 1;
}
```

Listing 5.1: Basic spin lock implementation

location it does what it needs with the data and then it must release it, it does this by storing the word back. This will write the data to the location and set the presence bit, allowing something else to acquire it. An implementation of this and example of using it to implement an atomic counter is given in listing 5.2. Whilst this could be simply emulated with a separate spin lock, acquiring the lock and then reading the value of the word must be done in two separate operations. **VL** allows these to be merged into a single operation.

5.1.3 Notify nodes and chains

A presence bit can be used as a fine grained notification mechanism. A thread that wishes to be notified of some event can **TAC** a word and then immediately load it and attempt to use the result. As the word has had its presence bit cleared the thread will pause. When the event the thread wishes to be notified about occurs the word is written to and the thread woken up. The word write could be used to pass some data to the thread as part of the notification or the result of the load may be discarded, if all that matters was the event occurring. This is known as waiting on a word, a sample implementation can be seen in listing 5.3.

This is a commonly used pattern, so a structure known as a notify node is introduced. This contains two words one, known as the wait word, is the word that has its presence unset and has a thread wait upon whilst awaiting some event. The other is a pointer to another notify node. Nodes can be linked together to form a notify chain. A notify chain is used when multiple threads wish to wait upon the same thing. When a thread wishes to wait on a notify chain it creates a notify node and adds it to the chain, it then waits on the wait word. A thread wakes the notify chain by writing to the first wait word, waking the first thread. The just-woken thread can then continue to wake the next node in the chain or it can perform some action before waking the next node (producing either a notify one or notify all behaviour).

The implementation of notify chains must atomically insert a notify node into the chain. It must also avoid a potential lost wakeup problem. Say there is a condition C, that if

```
//Acquire a location, this returns with the location's value and the location's presence
    will be cleared.
uint64_t acquire_location(uint64_t* location) {
        uint64_t got_exception;
        //location_value must be stored in a specific register as the exception status
            word (ESW) is checked for a read exception on that particular register
        register uint64_t location_value;
        do {
                //load the location's value with VL, if the location is present the
                    value will be returned and the location's presence cleared.  If the
                    location is not present a read exception will be returned which sets
                    a bit in the ESW corresponding to the VL's destination register
                location_value = VL(location);

                //The read exception will set a bit corresponding to the VL's
                    destination register in the ESW, check_ESW looks at this bit and
                    return its value
                got_exception = check_ESW(location_value);

                //If we got an exception we need to retry the VL, if we didn't we got
                    the value
        } while(got_exception);

        return location_value;
}

//Atomically increment the 64-bit word pointed to by counter
void atomic_increment(uint64_t* counter) {
        //First acquire the counter location, we get a local copy of the current counter
            value.  The 'master' value will be marked as non present, whilst it is non
            present nothing else can acquire it
        uint64_t current_counter = acquire_location(counter);

        //Increment our local value
        current_counter++;

        //Store it back, setting the presence of the counter location so something else
            is able to acquire it
        *counter = current_counter;
}
```

Listing 5.2: Acquiring a location and implementing an atomic counter

true allows a thread to continue but if false means a thread must wait on a notify chain which will be notified when C becomes true. Thread A checks C and finds it to be false so prepares a notify node to add to the chain, meanwhile thread B performs an action causing C to become true, so wakes the chain. Following this thread A atomically inserts itself into the chain, missing the notify from thread B and potentially waits on the chain forever.

Similar problems are encountered in the MCS lock [73] which is a type of spin lock that utilises a queue. When a thread wishes to enter the lock it prepares a new node for the queue. Exactly as a notify node this contains two words, one is a boolean the other is a pointer to the next node in the queue. The thread checks the queue, if it is empty nothing has the lock and thread may proceed (the thread acquires the lock). If it contains another node the thread adds its node to the queue and then sits in a busy waiting looping waiting for the boolean in its node to change from false to true. To release the lock a thread writes true into the boolean field of the node pointed to by the thread's node's next pointer.

Only a tail pointer for the queue is maintained, to atomically insert into the queue a compare and swap operation can be used to switch the tail from its current value to

```
//Causes a thread to wait until a word become present (or more specifically til its
    value is 0 and it is present), clears the word's presence initially. To wake up a
    thread waiting on a word like this use notify_word on that word
void wait_on_word(volatile uint64_t* word) {
        //Clear the presence bit of the word
        TAC(word);

        //Looks like a busy wait loop but the first load of the word will cause the
            thread to wait because the word is not present, when another thread wakes
            this thread by writing 0 to the wait word the presence bit will be set so
            the thread will wakeup, the while loop condition will be false so the loop
            exits.
        while(*word);
        //Doesn't have to be a while loop, could load and then use the result in an add
            with register zero as its destination register (register zero is a constant
            0 so using it as the destination of an add has the effect of discarding the
            result), the thread cannot execute the add until the word becomes present
            and a read response is returned.
}

//Wakes up a thread waiting on word
void notify_word(volatile uint64_t* word) {
        //Store 0 to the word which makes the word present and causes the while loop in
            wait_on_word to exit
        *word = 0;
}
```

Listing 5.3: Waiting on a word

the new node. Then the next pointer on the old tail can be updated to point to the new tail with a standard store. The lost wakeup occurs when a thread, A, releases the lock. Thread B may have created a new node and switched the tail pointer, but the next pointer of the old tail hasn't been updated yet, so when thread A checks the next pointer of its node to find something to wake it finds nothing. Thread B then finally changes the next pointer of the old tail to point to its node, but its too late, thread B will never exit its busy waiting loop. To prevent this thread A attempts to compare and swap the tail pointer from its node (if it's releasing the lock and nothing else is acquiring its node must be the tail) to NULL. If this fails then another thread is adding its node to the queue so thread A enters a busy waiting loop waiting for its node's next pointer to be updated.

The notify queue is very similar in structure and implementation to the MCS lock, but with two crucial differences, (i) it waits on a wait word rather than sitting in a busy waiting loop waiting for a boolean to become true, (ii) instead of using compare and swap to atomically insert nodes into the chain and to avoid the lost wakeup issue the tail pointer location is acquired with **VL** to perform atomic updates.

The lost wakeup issue as described above only occurs when we have a condition C that is linked to whether or not we wait on the notify chain and is altered by something that also notifies the chain. In the MCS lock this condition C is effectively, lock acquired or not. There may be situations where no condition exists, e.g., if the notify chain was used to notify threads of some external event (e.g., a timer expiring, or an IO event occurring) in which case a thread missing a notify is not an issue as another one will be triggered from elsewhere so it won't wait forever.

The precise condition which causes the lost wakeup issue depends upon the use the notify chain is put to (concrete examples are given below). In general anything notifying the notify chain or adding a node to the notify chain must first acquire the tail pointer before performing the action that alters the condition or checks the condition. After that is done the tail pointer is replaced so something else can acquire it. As only one thread can acquire the tail pointer at once the lost wakeup cannot occur as the condition cannot change between a thread deciding to add itself to the notify chain and the addition occurring. Effectively the acquisition of the tail pointer is acting as a lock, the lock must be acquired before the notify chain is used. A notify chain could also use a head pointer rather than a tail pointer, the mechanism is exactly the same apart from new nodes will be inserted at the front rather than the back.

In general waiting on a notify chain works as follows:

1. Create a notify node
2. Acquire the tail pointer (or head pointer if this notify chain wants insert at front behaviour).
3. Check the condition C, to see if a notify chain wait is needed.
4. If no wait needed, release tail pointer and continue.
5. Otherwise insert node into notify chain:

   Change next pointer of current tail to point to new node (or change next of new node to point to current head).

   Update tail/head pointer to point to new node (releasing tail/head pointer so another thread can acquire it).
6. Wait on wait word of notify node, if something has notified the chain between this step and updating the tail pointer then the wait word will be present and thread continues immediately.
7. After wait on wait word finishes, write to the wait word of the notify node pointed to by our node's next if notify all behaviour is desired.

To wake a notify chain simply write to the wait word of the head node of the notify chain having obtained the tail/head pointer first, then return the tail/head pointer.

Similar Mechanisms

The notify chain takes inspiration from the MCS lock [73] but also shares many similarities with other mechanisms. Three examples (discussed in the fine-grained computation chapter) are Tullsen's lockbox [100], the queue on lock bit mechanism (QOLB) [52] and the queue based locks from the DASH multiprocessor [64]. All three serve a similar purpose. They seek to reduce or entirely remove the excess memory accesses in a multicore system when several processors are spinning on a lock. In QOLB this is accomplished by any processor spinning on the lock taking a shadow copy of the cache line that holds it and spinning on that locally. In each cache stored with the cache lines is a pointer, these pointers can form a queue. When something releases a lock it only needs to notify the processor next in the queue rather than invalidating all shadow copies everywhere. DASH's queue based locks do something similar only instead the sharing directory from the cache coherency system, which knows what processors are sharing the lock's cache line, is used to determine where the lock should be acquired next. QOLB differs from notify chain as it requires extra data in each cache (compared to the single presence bit required for the notify chain), it also operates on a per-processor basis, the notify chain operates on a per-thread basis. DASH's queue based locks use existing state within the cache coherency system, but again it operates on a per-processor basis, not a per-thread basis.

Tullsen's lockbox takes a different approach, any thread spinning on a lock via a special acquire instruction can be stalled, with the address of the lock and acquire instruction placed in a lockbox (there is one per thread), when the lock is released, any lockbox that has the address of the lock is notified and its thread can be rewoken. This differs from the notify chain as it requires an extra per-thread hardware structure and it requires a general way to efficiently broadcast the releasing of a lock to any lockboxes which hold its address. Finally it should be noted that all three mechanisms require specific hardware support. The notify chain is built in software on the presence bit mechanism.

Notify chains are a general mechanism that can be used in many ways, below are three possibilities:

Locks

These follow directly from the implementation of a notify chain being taken from the MCS lock implementation. To acquire the lock follow the general process for waiting on a notify chain above with the following changes:

> Condition C is whether or not tail is NULL. If the tail is NULL nothing has the lock so the thread can acquire it immediately
>
> If the tail is NULL a thread must set the tail to its notify node when acquiring the lock. This removes the need for a separate head pointer for the chain.
>
> If the tail is not NULL then wait on the wait word of the notify node. When awoken don't write to the next node in the chain as notify one, not notify all behaviour is desired

Once a thread is done waiting on the wait word of its notify node (or if it never had to wait because the tail pointer was NULL) it has acquired the lock. To release it it writes to the wait word of the notify node pointed to by the next field of its notify node. If its notify node has a non NULL next field then it can just write directly to the wait word of that node and its done. If its notify node has a NULL next field then currently there is no next thread to wakeup, however it is possible that another thread has determined the lock is acquired but hasn't added itself to the notify chain yet. So to avoid a lost wakeup it acquires the tail pointer, then if the tail pointer points to the notify node of the thread releasing the lock nothing else has been added to the chain so it may safely set the tail to 0 and do nothing else. Otherwise the tail has changed so there is another thread to

```
//Acquires the notify chain lock lock, also needs a notify node to add to the notify
    chain
void acquire_lock(volatile notifyChainLock_t* lock, volatile notifyNode_t* notifyNode) {
        //To wait on the notify node we need to clear its wait word
        TAC(&notifyNode->waitWord);

        //If adding notify node to the chain it will be at the tail so set next to NULL
        notifyNode->next = 0;

        //Acquire the tail pointer
        notifyNode_t* notifyChainTail = acquireLocation(&lock->chainTail);

        //If tail is NULL, nothing has the lock so add notify node as the only node in
            the chain and return immediately because the lock has been acquired.
        if(notifyChainTail == 0) {
                //Make wait word present again as we're not going to wait on it
                notifyNode->waitWord = 0;
                lock->chainTail = notifyNode;
                return;
        }

        //Otherwise tail is not NULL, something else has the lock, add the notify node
            to the tail
        notifyChainTail->next = notifyNode;
        lock->queueTail = notifyNode;

        //Wait for the wait word to become present (looks like a busy waiting loop but
            will cause the thread to sleep until notifyNode->waitWord becomes present at
            which point the loop will exit as we'll write 0 to notifyNode->waitWord)
        while(notifyNode->waitWord);
}
```

Listing 5.4: Notify chain lock acquire implementation

wakeup so it writes to that thread's notify node's wait word. The code to acquire the lock can be seen in listing 5.4 and the code to release the lock can be seen in listing 5.5.

Barriers

Any thread waiting on a barrier will wait until the number of threads waiting on that barrier reaches some value, at which point they will all be notified and all threads wake and continue execution. Along with a notify chain they require two integers, one is the total thread count required and the other is the number of threads currently waiting on the barrier. To wait on the barrier follow the general process for waiting on a notify chain above with the following changes:

> Condition C is whether the number of threads waiting is equal to the total number of threads we're waiting for. If the thread entering the barrier will bring the number of threads waiting up to the total we're waiting for then that thread can immediately continue, it must also wake the notify chain and reset the number of threads waiting to 0. The number of threads waiting must be checked after the head pointer has been acquired. If it is checked before then multiple threads could observe the same value, this is not an issue for a thread that will be waiting on the chain for the barrier to complete (provided the count is incremented atomically), but if multiple threads read the same value and those threads are the last to wait on the barrier then one of them must wake the chain and only one of them must wake the chain

```
//Release the notify chain lock lock, must supply the same notify node that was used for
    acquiring the lock
void release_lock(volatile notifyChainLock_t* lock, volatile notifyNode_t* notifyNode) {
        //If the notify node has a non NULL next then that's the next thing that should
            acquire the lock
        if(notifyNode->next) {
                //So wake it
                notifyNode->next->waitWord = 0;
                //And we're done
                return;
        }

        //Otherwise either nothing is attempting to enter the lock, or something has
            seen the lock is acquired but hasn't been added to the notify chain yet, so
            acquire the tail pointer location
        notifyNode_t* notifyChainTail = acquireLocation(&lock->chainTail);

        //If tail is still the supplied notify node then nothing has been added to the
            notify chain
        if(notifyChainTail == notifyNode) {
                //As only one thread can have acquired the tail at once we can be sure
                    nothing is about to add itself having already looked at the tail, so
                    replace tail with NULL and we're done
                lock->chainTail = 0;
        } else {
                //Otherwise tail has changed so there is now something to wakeup so put
                    the tail back
                lock->chainTail = notifyChainTail;
                //Wakeup whatever is next in the chain
                notifyNode->next->waitWord = 1;
        }
}
```

Listing 5.5: Notify chain lock release implementation

so it is vital that precisely one of them see a number of thread waiting one less than the total expected. All threads must acquire the head pointer in any case because they are either adding themselves to the chain or waking the chain.

If the thread isn't the last in and must wait on the notify chain then it increments the threads waiting counter.

After a thread is done waiting on the notify node it must notify the next node in the chain as pointed to by its node's next pointer as notify all behaviour is desired.

The barrier uses a head rather than tail pointer as the last thread in needs to wake the head of the chain, if only a tail pointer is used (like in the lock implementation) then it doesn't know what the head is. This does result in a last in, first out behaviour for the barrier but as it is notify all this isn't a major issue. Both a head and tail pointer could be maintained if first in, first out behaviour is required.

The threads waiting count gets reset by the last thread in to the barrier which also wakes the chain. It resets the count before waking the chain which enables a thread to immediately wait on the barrier again even if other threads have yet to be awoken (i.e. they're still waiting on the barrier). This will not cause issues because a thread that hasn't been awoken yet doesn't need to check the thread waiting count (so arbitrary changes it to are fine) and the notify chain it is in won't be disturbed by another thread waiting on the barrier again. The barrier implementation can be seen in listing 5.6

```
//Wait on barrier b, notifyNode used when thread must wait (i.e. it's not the last
    thread to call waitOnBarrier)
int waitOnBarrier(barrier_t* b, notifyNode_t* notifyNode) {
        //Acquire head of barrier's notify chain
        notifyNode_t* chainHead;
        chainHead = acquireLocation(&b->notifyChain->head);

        //If we're the last thread in
        if(b->threadsWaiting + 1 == b->totalWanted) {
                //Wake notify chain, if there is anything in the notify chain
                if(b->notifyChain->head)
                        b->notifyChain->head->waitWord = 0;

                //Reset thread waiting counter, as well as head of notify chain so
                    barrier can be immediately reused (threads being woken aren't
                    effected by changes in these values)
                b->threadWaiting = 0;
                //Head reset last as that puts head pointer back, allowing other threads
                    to wait on the barrier again.
                b->notifyChain->head = 0;
        } else {
                //We're not the last thread in, so increment threads waiting count (we
                    have acquired the head pointer, so safe to increment this as no
                    other threads can be here at the same time)
                b->threadsWaiting++;

                //Prepare notify node for use
                TAC(&notifyNode->waitWord);
                notifyNode->next = 0;

                //If the notify chain is empty
                if(chainHead == 0) {
                        b->notifyChain->head = notifyNode;
                } else {
                        //Otherwise just add to front and set head appropriately
                        notifyNode->next = chainHead
                        b->notifyChain->head = notifyNode;
                }


                //Wait for wakeup
                while(notifyNode->waitWord);

                //We want to notify all so if we have a non NULL next pointer on our
                    notify node wake that also
                if(notifyNode->next)
                        notifyNode->waitWord = 0;
                }
        }
}
```

Listing 5.6: Barrier implementation

Single producer   multiple consumer

The Mamba architecture inherently supports a single producer   single consumer communication style. Initially the location a producer and consumer wish to use to communicate is set to non-present, then the consumers loads it. The consumer thread will either continue immediately because the producer has already stored to it (so it has become present) or it will wait until the producer stores to it (because it is still non-present). If a second consumer loads the location when it is non-present then it will receive a read exception (as the first consumer will have already setup a forward address in the non-present location when it did its load).

A notify chain can be used in this situation, if a consumer has already attempted a load of a non-present location (setting up a forwarding address at that location so anything else will receive a read exception on a load) then further consumers can join a notify chain which gets notified by the producer when that word gets written.

A consumer should use the **LDNR** rather than **LD** instruction as it is expecting a read exception might occur. After performing the **LDNR** the ESW is checked, if the load succeeded than the consumer continues, otherwise it joins the notify chain. It is important a consumer performs an **LDNR** before trying to wait on the chain. If the first one is done after acquiring the head pointer of the notify chain and the location is not-present with no forwarding address the thread will pause until the location becomes present whilst it still has the head pointer acquired preventing anything else from waiting on the chain. It uses the general process for waiting on a notify chain above with the following changes:

> Condition C is whether the **LDNR** succeeds or not (that is whether the ESW indicates an exception or not), if it does succeed location is present and no wait required. For the first consumer **LDNR** will succeed (no read exception will occur) but not immediately, the thread will wait until the producer writes to the location. Still no interaction with the notify chain is required as the wait has been handled by the architecture.
>
> Otherwise **LDNR** does not succeed so wait on the notify chain. When a producer produces a value it should wake the chain by storing that value into the wait word of the head node of the chain. So when a thread is woken by its node's wait word being stored to it has been given a copy of the value, avoiding the need to load the (now present) word being consumed again. When a thread is woken it should store the value loaded from its wait word into the wait word of the next node in the chain as a notify all behaviour is desired

Again a head pointer is used rather than a tail pointer as the producer needs to know what node is the head of the chain. When it produces a value the producer first stores it to the word being consumed then it must acquire the head pointer of the notify chain before writing the produced value to the wait word of the head node of the chain. Otherwise a lost wakeup issue is introduced (a consumer thread may have failed an **LDNR** but not added itself to the chain when the producer thread wakes it so the consumer thread waits forever).

5.1.4 FIFO communication

Lamport presents a implementation of a single producer single consumer FIFO based around a ring buffer in [58]. Along with the ring buffer, read and write pointers are maintained. Given a ring buffer size b these are stored as integers modulo $2b$, so the value of the pointer modulo b gives the slot in the ring buffer it is pointing at. The read pointer points to the next slot in the ring buffer that the producer should read from, the write pointer points to the next slot in the ring buffer that the consumer should write to. The empty condition is indicated by the read and write pointer pointing to the same spot in

```
//Attempt to load a value from location waitLocation with LDNR, if location is
    non-present and has a forwarding address thread will wait on notify chain 'chain'
    using notify node notifyNode. Any producer storing to the location should wake
    chain with wakeChainWithValue. Function will return the value at the location,
    either immediately because the location is present or after it has woken by producer
    storing to the location and waking the chain with wakeChainWithValue.
uint64_t waitForValue(volatile notifyChain_t* chain, volatile notifyNode_t* notifyNode,
    volatile uint64_t* waitLocation) {
        //Acquire chain head
        notifyNode_t* chainHead;
        chainHead = acquireLocation(&chain->head);

        //Attempt an LDNR on the location, at this point we assume an LDNR has already
            been tried and given an exception, need to try after acquiring head to avoid
            lost wakeup
        uint64_t locationValue;
        uint64_t gotException = 0;
        locationValue = LDNR(waitLocation);
        gotException = check_ESW(location_value);

        //If we have no exception on LDNR then between first LDNR and this one producer
            has stored to the location
        if(!gotException) {
                //So put head back and return the location value we got
                chain->head = chainHead;
                return locationValue;
        }

        //Otherwise wait on chain, so prepare notify node for use
        TAC(&notifyNode->waitWord);
        notifyNode->next = 0;

        //If the notify chain is empty
        if(chainHead == 0) {
                //Just need to set the head
                notifyChain->head = notifyNode;
        } else {
                //Otherwise add to front and set head appropriately
                notifyNode->next = chainHead;
                notifyChain->head = notifyNode;
        }

        //Read the value and pass it on to the next node in the chain immediately as
            notify all is desired. The presence bit ensures that this won't occur until
            the value has actually been produced.
        uint64_t value = notifyNode->waitWord;
        if(notifyNode->next)
                notifyNode->next->waitWord = value;

        return value;
}
```

Listing 5.7: Single Producer Multiple Consumer wait for value implementation

the buffer. The full condition is indicated by the distance between the two pointers being equal to the size of the ring buffer (this will be different to them both pointing to the same slot because they are stored modulo $2b$ rather than modulo b).

A producer checks if the buffer is full, if it isn't it writes the word it is producing to where the write pointer points and then increments it (storing the result back modulo $2b$). If the buffer is full the producer must sit in a busy waiting loop waiting for the read pointer to change (indicating that the consumer has taken a word out of the FIFO so a slot is now free for the producer to write to) or perform some other work.

```
//Wake a notify chain, chain with a value, value.
void wakeChainWithValue(notifyChain_t* chain, uint64_t value) {
        //Acquire head first
        notifyNode_t* chainHead;
        chainHail = acquireLocation(&chain->head);

        //If there's something in the chain
        if(chain->head)
                //Then write the value to its wait word. Each thread waiting on the
                    chain will take care of writing the value to the next node in the
                    chain
                chain->head->waitWord = value;

        //Clear the chain for reuse
        chain->head = 0;
}
```

Listing 5.8: Single Producer Multiple Consumer wake chain with value implementation

A consumer checks if the buffer is empty, if it isn't it reads the word pointed to by the read pointer and then increments it (storing the result back modulo $2b$). If the buffer is empty the consumer must sit in a busy waiting loop waiting for the write pointer to change (indicating that the producer has put a word into the FIFO that can now be consumed) or perform some other work.

Note that no locks or special atomic operations (such as compare and swap) are needed for this to function correctly. This is because whilst both producer and consumer examine both pointers only the consumer updates the read pointer and only the producer updates the write pointer. Updating the write and read pointers after a word has been produced or consumer respectively prevent race conditions between the producer and consumer both acting on the same word of the ring buffer at the same time. If multiple producers or consumers are introduced the algorithm is not sufficient to ensure correctness.

Using the facilities of the Mamba architecture we can improve upon this to easily remove the busy waiting required by both the producer and consumer. First to remove the busy waiting required of the consumer the ring buffer starts with all of its slots marked as non-present. The check for an empty buffer is removed from the consumer routine so it immediately loads from the slot pointed to by the read pointer. If the producer hasn't produced anything (i.e. the buffer is empty and the read pointer is pointing at an empty slot) then the load will be targeted at a non-present word so the consumer will wait until the producer writes something there. Upon successfully completing a load the consumer immediately clears the presence bit of the word just loaded from before incrementing the read pointer (whilst the read pointer remains where it is the producer thread won't write to the slot it points to so it is safe to clear the presence bit). So any word that has yet to be consumed has the presence bit set, otherwise it will be unset so the consumer can safely load the read pointer directly without worrying about empty/not-empty because if it is empty the consumer will automatically wait until the producer writes to the buffer. The consumer implementation can be seen in listing 5.9.

Removing the busy wait loop from the producer is slightly more involved. After the producer has determined the ring buffer to be full it needs to be able to wait for the consumer to consumer something, without continuously checking the full/not-full state of the buffer. This is done via a single word which the producer can wait on when the buffer

```
//Consume a word from FIFO fifo, consumed data is stored to the word d points to
void fifo_consume(volatile uint64_t* d, volatile FIFO_t* fifo) {
        //Load from buffer slot pointed to by the read pointer (modulo fifo size), no
            need to check if there's anything in the FIFO as if there isn't the slot
            won't be present and the consumer thread will wait until the producer writes
            something to the FIFO (making the slot the consumer attempted to load from
            present).
        *d = fifo->buffer[fifo->readPointer & (fifo->size - 1)];
        //As soon as we've loaded the data from the slot, clear its presence bit
        TAC(&fifo->buffer[fifo->readPointer & (fifo->size - 1)]);

        //Increment read pointer modulo 2 * fifo_size (fifo->modMask == (fifo->size * 2)
            - 1, so & performs the modulus provided the fifo size is a power of two)
        fifo->readPointer = (fifo->readPointer + 1) & fifo->modMask;
        //Set the presence of the producer wait word waking up the producer if it is
            waiting for the FIFO to become non-full
        fifo->producerWait = 0;
}
```

Listing 5.9: Implementation of FIFO consumer without busy wait loop

```
//Puts a produced word d into the FIFO fifo
void fifo_produce(uint64_t d, volatile FIFO_t* fifo) {
        //First clear the producer wait word
        fifo->producerWait = 1;
        TAC(&fifo->producerWait);

        //Check to see if fifo is full
        if(((fifo->writePointer - fifo->readPointer) & fifo->modMask) == fifo->size) {
                //If so we wait on the producer wait word
                while(fifo->producerWait);
        }

        //Store to the buffer at the slot pointed to by the write pointer, at this
            point, either the fifo wasn't full in the initial check, or the producer has
            been waiting on the producer wait word and has been woken by the consumer
            consuming so there's no space to store.
        fifo->buffer[fifo->writePointer & (fifo->size - 1)] = d;
        //Increment write pointer modulo 2 * fifo_size (fifo->modMask == (fifo->size *
            2) - 1, so & performs the modulus provided the fifo size is a power of two)
        fifo->writePointer = (fifo->writePointer + 1) & fifo->modMask;
}
```

Listing 5.10: Implementation of FIFO producer without busy wait loop

is full. Before checking if the buffer is full or not the producer clears the presence of the word (known as the producer wait word), if then checks to see if the buffer is full, if it is it waits on the producer wait word. When the wait is done it is clear to write to the write pointer. The consumer stores to the producer wait word after it has updated the read pointer every time it consumes, if the producer is waiting for the buffer to become not-full this will wake it up. The producer implementation can be seen in listing 5.10.

After the producer has performed the **TAC** that clears the presence bit of the producer wait word there are a few possibilities:

1. The buffer wasn't full, so after examining the read and write pointers the producer determines its clear to store a word in the buffer. The consumer can only alter the read pointer and cannot make the buffer become full so it is safe for the producer to store to the buffer regardless of what the consumer does between the producer clearing the producer wait word and deciding to store to the buffer.

2. The buffer is full, so after examining the read and write pointers the producer determines it must wait so it waits on the producer wait word, the following things may occur:

   (a) Between the presence bit of the producer wait word being cleared and the producer loading the wait word the consumer hasn't done anything, so the producer wait word remains non-present. The producer will eventually be woken by the consumer writing to the producer wait word after it has consumed something and altered the read pointer so it is safe for the producer to continue and store to the buffer.

   (b) Between the presence bit of the producer wait word being cleared and the producer loading the wait word the consumer consumes. The consumption involves two steps that are visible to the producer, the updating of the read pointer and the setting of the producer wait word presence bit. If the read pointer is updated before the producer examines it then the producer will (correctly) determine the buffer isn't full and immediately continue with writing to a spare slot. If the read pointer is updated after the producer examines it then the producer will wait upon the producer wait word. At some point the consumer will store to the producer wait word, setting its presence bit, if this occurs before the producer starts the wait, then when it does (by loading the producer wait word) the producer will immediately continue and can safely write to the ring buffer. If the consumer stores to the producer wait word after the producer starts the wait then the producer will be woken up, continue and can safely write to the ring buffer. There is no possibility of a lost wakeup as only the producer clears the presence bit of the producer wait word and there is only one producer so the bit won't be set and cleared again between the producer deciding to wait and the wait actually beginning.

So the producer will successfully write a word to the FIFO without having to continuously poll. The word will either be written immediately because there was space or after waiting and being woken by the consumer.

5.2 A SAT solver implementation

5.2.1 SAT problem

The SAT problem is one of deciding whether or not a particular boolean formula can be satisfied, that is there is an assignment of booleans to the free variables in the formula that makes that formula true. The SAT problem is of interest for a few reasons. Firstly many things can be constructed as a SAT problem. A common example would be formal verification of hardware. A SAT instance can be constructed that is satisfiable if and only if a given hardware design meets certain criteria. Secondly SAT is NP complete so it is non-trivial to write a useful SAT solver so their development is an area of active research. Finally it is a problem amenable to parallelization but it is not straight forward, intricate synchronisation is required.

An SAT problem can be represented in conjunctive normal form (CNF). A CNF formula consists of a number of clauses. Each clause is a disjunction (OR) of variables (which may be negated) and the formula is a conjunction (AND) of clauses. Any boolean formula can be transformed into CNF so it can be used as a general representation. An advantage of CNF is it gives a regular structure to any particular instance of the SAT problem. A small example can be seen in equation (5.1).

$$(A \quad B \qquad C) \quad (A \qquad B \qquad C) \quad (\quad A \quad B \qquad C) \tag{5.1}$$

A satisfying assignment is one that makes the formula true, in this case a satisfying assignment would be $A = T$, $B = T$, $C = F$ where T is true and F is false.

The SAT solver presented here is not intended to compete with modern SAT solvers. It exists as a test case for the architecture. It is not essential that is compares favourably with other SAT solvers provided it exposes enough parallelism that a useful conclusion can be drawn about Mamba from it.

5.2.2 SAT solver algorithm

An obvious way to implemented a SAT solver is via an exhaustive backtracking search. Variables are assigned in turn until a contradiction is reached (the partial assignment produces a false result) at which point the solver backtracks and tries a different assignment. The method used by the solver is taken from that of Davis-Putnam-Logemann-Loveland (DPLL)[24]. The DPLL method proceeds as backtracking search does, choosing a variable to assign and then assigning it. When a variable is assigned all clauses are searched for ones containing the assigned variable. If the assignment makes the clause true (trivial to determine as the clause is an OR of variables) the clause is removed. If the assignment is to a variable in the clause but doesn't make it true (i.e. assignment is to true but variable is negated in the clause or vice versa) the variable is removed from the clause. If an empty clause is created the current partial assignment produces a contradiction so backtracking is required. If all clauses are removed then the formula has been satisfied. Along with the basic search DPLL adds the following inferences:

Pure Literal Elimination If every occurrence of a particular variable in the remaining clauses is negated or non-negated then that variable can be instantly assigned.

Unit propagation If a clause only contains a single variable then in order to make the formula true there is only one possible assignment for that variable (the one that makes the clause true).

If either inference determines a particular assignment must be made this will be done before choosing the next variable to assign, this often leads to further inferences. This has the effect of removing large parts of the search space that the backtracking algorithm without any inference would explore.

The SAT solver used in the evaluation only uses the unit propagation inference.

5.2.3 Parallelizing the algorithm

This algorithm is parallelized by splitting up the clauses of the problem between a given number of threads, known as the process threads. A single control thread directs the process threads. It chooses which variable to assign next and what to assign it to. This decision is relayed to all of the process threads which attempt the assignment on all of their assigned clauses. If it results in a contradiction the control threads tells all process threads to backtrack. If it doesn't each process thread prepares a list of unit inferences (i.e. assignments that must be made due to unit propagation discussed above) which are fed back to the control thread. It checks these inferences do not contradict, backtracking if they do, and sending them to all process threads if not.

Presence bits can be used to coordinate the process threads. Each process thread has an action (assign or backtrack) and an assignment (i.e. a variable name and true or false) and produces a result (whether or not that assignment produced a contradiction). For every process thread there is an action word and a result word which start non present. The thread reads the action word, clears its presence bit and then performs the action. The control thread writes an action to every process thread's action word and then reads the result word, clearing its presence bit after. The presence bits ensure that a process thread only performs an action once the control thread has given it one and that the control thread may only continue once all process threads have returned a result. See listing 5.11 for pseudo-code of the top level loops.

When a process thread performs an assignment it may generate unit inferences, these are checked as the process thread performs the assignment on all of it's clauses to see if they conflict. If they do not they must be propagated back to the control thread, this is done via a FIFO. So when the control thread has assigned a variable if all process threads have completed the assignment without causing a contradiction it reads all of the unit inferences from all of the process threads' FIFOs. These are checked for contradictions, causing the control thread to issue a backtrack if one is found or further assignments if none are found.

The MIPS64 implementation is identical to the Mamba implementation, apart from the control and process thread coordination. Instead of a process thread waiting for an action word to become present it instead continuously polls the action word until it becomes a valid action. Similarly the control thread after issuing actions polls the result word for each process thread until it becomes a valid result. FIFOs are still used to propagate unit inferences (albeit with different internal implementations).

A clear bottleneck in the presented parallelization is the single control thread. Though this allows the study of how Mamba behaves when an applications scaling is limited by its implementation. Furthermore as shown in the evaluation a fair speedup is achievable before the bottleneck limits performance.

5.3 Summary

This chapter discusses how software can be built for Mamba, the basic concept of waiting on a word is introduced which is used to create a notify chain. A simple and general queue

```
process_thread() {
        while(True) {
                action = *(action word)
                switch(action) {
                        case ACTION_ASSIGN:
                                TAC(action word)
                                \\do assignment..
                                *(result word) = result
                                break
                        case ACTION_BACKTRACK:
                                TAC(action word)
                                \\do backtrack...
                                *(result word) = result
                                break
                }
        }
}

control_thread() {
        while(True) {
                action = decide_action()
                foreach(process thread) {
                        TAC(result word)
                        *(action word) = action
                }

                result = RESULT_OK
                foreach(process_thread) {
                        if(*(result word) == RESULT_FAIL)
                                result = RESULT_FAIL
                }

                process_result(result)
        }
}
```

Listing 5.11: SAT Solver process and control thread loop implementation

based mechanism for a number of threads to efficiently wait for an event to occur that can be utilised to implement locks, barriers and a single producer multiple consumer communication pattern. The concept of waiting on a word is also used to implement a single producer single consumer FIFO.

Chapter 6

Evaluation

6.1 Methodology

The evaluation of Mamba aims to show that the new architectural features added by Mamba, namely the use of a tagged memory combined with a hardware scheduler, are effective in enabling the use of fine-grained threading techniques. To show this Mamba must be capable of keeping synchronisation and scheduling overheads low as thread and core count is increased allowing good scaling. To this end Mamba is compared to a similar MIPS64 based system (described below) that lacks a tagged memory and hardware scheduler. Three microbenchmarks were constructed, one based around locking, one based around barriers and one based around FIFO communication. Mamba is found to exhibit better scaling than the MIPS64 system as well as being insensitive to thread count, in situations where the performance of the MIPS64 systems rapidly degrades with increasing thread count Mamba maintains its performance.

The SAT solver described in the software techniques section is evaluated as an example of a full application. It is run using varying numbers of process threads and cores with two different SAT problems. The Mamba implementation is found to perform better than the MIPS64 implementation as well as degrading more gracefully at extremes of thread count. When the thread and core count capable of reaching maximum performance (where further scaling is limited not by the hardware but by the algorithm employed) is reached Mamba is more able to maintains this maximum performance as threads and cores are increased compared to the MIPS64 implementation before overheads start to degrade this performance.

6.2 MIPS64 comparison system

The MIPS64 comparison system is very similar to Mamba. Like Mamba is executes a subset of the MIPS64 ISA, however it doesn't include the extensions added by Mamba. It uses a 5-stage pipeline very similar to the original MIPS design. It has 8 hardware contexts as Mamba does, each cycle an instruction from a different context is fetched and enters the pipeline and the contexts are scheduled in a strict round robin order. Unlike Mamba there is no hardware scheduling system that chooses to switch contexts, all scheduling is handled by the software. An exception handling mechanism is included so a pre-emptive software scheduler can be used.

The MIPS64 core runs in the exact same system Mamba does, the interconnection network, caches and memory controllers are identical. MIPS64 uses the same caching system as Mamba in that each node in the systems owns an area of address space, addresses within that space are cached only by that node. An access by a node to an area of memory outside of its part of address space will always go via the cache that owns that part of address space.

A compare and swap (CAS) instruction is added to the MIPS64 ISA as an atomic primitive. The MIPS64 ISA includes linked load and store conditional instructions but these are usually implemented by utilising a cache coherency protocol. The caching structure used by Mamba and MIPS64 is not conducive to efficient implementation of these primitives which is why CAS was used instead.

6.2.1 The software scheduler

The MIPS64 system utilises a software scheduler supporting an unlimited number of software threads. It schedules threads in the exact same manner as the Mamba hardware scheduler, there is a single ready queue per core, when a thread's quantum expires it gets swapped out of its hardware context and a new thread swapped in in its place. The one difference is when threads get added to the ready queue, Mamba has the benefit of presence bits so only adds a thread to the ready queue when all of its registers are present. MIPS64 lacks this mechanism so a thread that is swapped out is added immediately to the back of the ready queue. An interrupt mechanism periodically triggers an exception in the MIPS64 system that allows the thread scheduler to switch threads. This occurs in every context (the scheduler is not capable of choosing to switch a thread into another context, it can only switch threads in the context it is currently running in). The scheduler is entirely disabled in any of the benchmarks below that use 8 threads or less.

Quantums are handled differently in Mamba and MIPS64. In both Mamba and MIPS64 a separate counter is associated with each hardware context which holds that context's quantum. In Mamba this is decremented every time an instruction for the context reaches the end of the pipeline (regardless of if it managed to complete or not, e.g., if an instruction accessed a currently non-present register the instruction would be repeated but the quantum counter would still be reduced by 1). In MIPS64 every quantum counter is decremented every cycle (in effect they are simple count-down timers).

MIPS64 also has a far larger quantum than Mamba. The quantum used for MIPS64 was chosen experimentally by examining the effect of the quantum on the microbenchmarks and choosing one which provided the best performance. The MIPS64 quantum is set at 200'000 cycles. As the MIPS64 quantum is decreased every cycle rather than every instruction through the pipeline like Mamba this means a context will perform a thread switch every $\frac{200000}{8} = 25000$ instructions if no instruction stalls and all hardware contexts are in use. The Mamba quantum is set at 1'000. So MIPS64 will switch 25x less often than Mamba. However a Mamba switch only occurs if there is a thread in the ready queue and the MIPS64 software has the ability to yield, which causes the scheduler to immediately deschedule the thread and schedule something else in its place so switching frequency won't always be 25x less in MIPS64. A smaller quantum for MIPS64 that is more comparable to Mamba was trialled but it was found that this caused awful performance. Only limited experimentation was conducted with the Mamba quantum as it is hardwired into the scheduler.

The MIPS64 context switch itself is very lightweight. When a quantum expires the thread jumps to an exception handler. This saves all of the current register values into memory, removes a ready thread from ready queue (there is one per core), adds the current thread to the back of the ready queue and finally restores the register values of the thread just removed from the ready queue. This is a sequence of around 150 instructions. As the ready queue is shared amongst all hardware contexts on a core it is protected by a simple spin lock. This lock only needs to be held during the dequeue and enqueue operation and as this is very simple this does not cause a small performance bottleneck, two threads would have to jump into the scheduler at very similar times (within a few cycles of each other) to contend and even if they did one would not have to wait very long for the other. Li et al. measured the time to took to perform a context switch under linux on a dual core 2GHz Intel Xeon processor [65]. They measured a time of 3.8 μs for a direct context switch (that is the context switch alone, without any penalties incurred due to cache misses etc.because of a change in working set) giving 7'600 cycles for a switch. MIPS64 is significantly quicker than this because its threading and scheduling model do not contain the complexities a modern operating system entails.

During a Mamba context switch all 32 registers of a context must be copied into memory and the 32 registers of the newly scheduled activation frame must be copied from memory into the register file. This process is handled by the context switch sequencer and does not involve the context. During the switch the context being switched is entirely removed from the pipeline. In order to avoid race conditions that may occur should a memory action targeted at the AFs being switched enter the core during the switch all memory actions are blocked from entering the core during the switch. This means that if another context executes a load or store operation during the switch the entire pipeline will stall until the switch is over.

6.3 Software tool chain

All benchmarks were written in C with a small amount of assembler used for initialisation and exception handling code with inline assembly used to access the new Mamba instructions within C code. The GNU assembler, GAS, had its MIPS backend altered to include the extra Mamba instructions. Clang and LLVM were used to compile C. When the implementation of the software was started LLVM only supported MIPS32. The author wrote a new MIPS64 backend for LLVM based on the existing MIPS32 backend. The exact same compiler tool chain was used to build software for both Mamba and MIPS64. The -O1 level of optimisation was used as higher levels were found to occasionally produce incorrect code.

The SCons build system was used to automate the building of software. As each core has

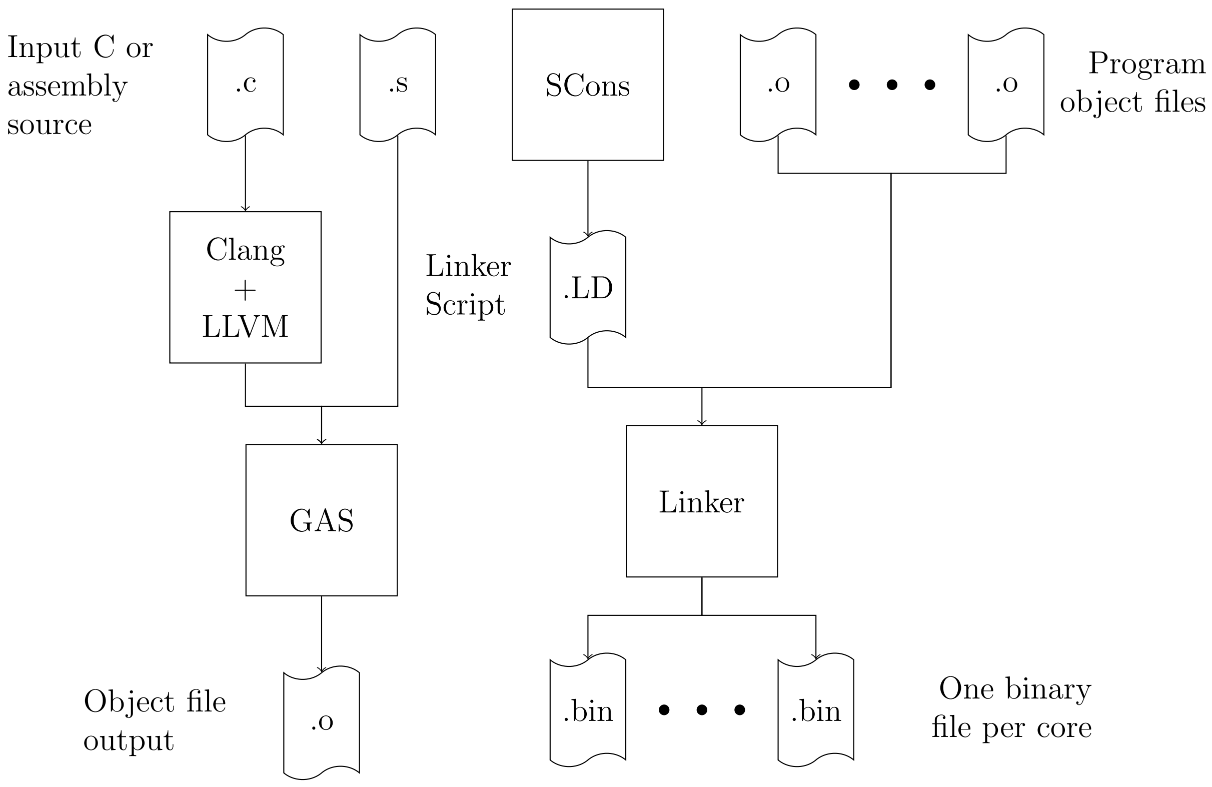


Figure 6.1: The Software Tool Chain. SCons orchestrates the entire process as well as generating linker scripts

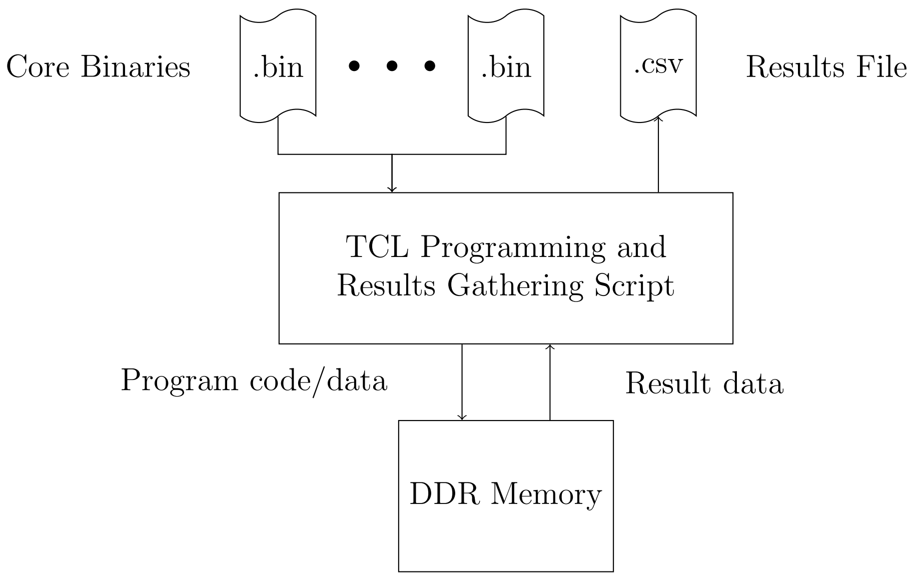


Figure 6.2: The program loading and running process

Off-chip memory access latency	9 cycles
Off-chip memory throughput	256 bits / cycle
Board Board serial latency	6 cycles
Board Board serial throughput	24 bits / cycle
Router Router on-chip latency	4 cycles
Router Router on-chip throughput	72 bits / cycle
Core Router latency	3 cycles
Cache access time	1 cycle
Cache miss penalty	11 cycles

Table 6.1: Performance of network and memory

a different local memory area every program had to be separately linked for each core to ensure addresses were correct. SCons automated the creation of the necessary linker scripts. Initially to load the programs one of the ethernet ports on the DE4 board was used. This required the porting of the lwIP TCP/IP stack to the Mamba and MIPS64 architectures. This provided a useful test case for the compiler tool chain. The network bootloader was stored in a ROM on the FPGA that was loaded into the DDR2 memory on power up. Eventually the use of this bootloader was discontinued to simplify the automation of loading benchmarks and gathering results. In its place an Altera provided component was used that allows direct access to the Avalon bus that connected the DDR2 memory to the MIPS64 and Mamba cores via the DE4's USB port. A TCL script automated writing a program into memory and taking the cores out of reset. When a benchmark was completed it wrote its results into a particular area of memory that the TCL script could then read and save the results into a CSV file. Figure 6.1 illustrates the software build and figure 6.2 illustrates the loading and running of a program on the FPGA.

6.4 Network and Memory performance

Table 6.1 gives various figures for the performance of the network and memory. It is notable that the off-chip memory latency is very small compared to any modern system, this is due to the 25 MHz clock rate of the FPGA system. However as the working set size of the benchmarks presented in this evaluation is small the off-chip memory latency is not a big factor in their performance.

The router   router figures given are for a flit that travels from the input port of one router, over an on-chip link and to the output port of another router. Effectively they are the latency and throughput of a single core   core network link. Any flit that must travel to a node on the other board incurs an extra penalty for traveling over the serial link. The core   router latency figure is the number of cycles a flit generated in a core takes to reach the input port of its tile router. Or from a flit from that router to reach the core. From this we can calculate the first flit of a remote memory request going to a neighbouring core will have a latency of $3+4+3 = 9$ cycles to go from being generated in one core's execute stage to reaching the other core's local memory where it will generate

Message type	Size in bits
MIPS64 Read Request	44
MIPS64 Write Request	95
MIPS64 Read/CAS Response	67
MIPS64 CAS Request	170
Mamba All Messages	134

Table 6.2: Network message sizes

Core	ALMs	Memory Block Bits (Total) Kb	Memory Block Bits (Cache) Kb	Memory Block Bits (Non-Cache) Kb
Mamba	15900	381.3	346.8	34.5
MIPS64	13200	311.5	308.0	3.5
Overhead	20%	22%	12%	890%

Table 6.3: Mamba and MIPS64 core resource utilisation

a response. The TILE64 architecture is reported to have total latency of 9 cycles and throughput of 31.44 bits per cycle over its networks [102]. Though the throughput given is in actual data transferred and the Mamba figure given is in raw bits.

6.4.1 Network Message Size

The network uses flits with a payload size of 58 bits, when including the header this gives a total flit size of 72 bits (this size was chosen as it matches the size of flits sent by the serial link). Whilst Mamba and MIPS64 use an identical network the size of messages sent differs between them. In Mamba all messages sent are 134 bits whilst in MIPS64 messages come in 4 separate categories each with a different size. These are given in table 6.2. It should be noted that no particular effort was made to optimise the messages sizes. Though a request in Mamba will always require more information than a request in MIPS64 as in Mamba any request has a response address referring to a particular word in memory, whilst in MIPS64 the request merely needs to state which node and context the response should be sent back to.

6.5 Area and memory utilisation

The area and memory utilisation for both Mamba and MIPS64 are presented in table 6.3. It should be emphasised that no time was spent on resource optimisation so the figures are included to give an idea of the relative cost of a Mamba core vs a MIPS64 core but do

[1]These figures are rounded to the nearest 100 and in the case of the DDR2 controller several figures were produced by the synthesis tool, one for each controller, that differed by up to 150 ALMs, the average of these was used and rounded. So whilst the area figures presented are identical the precise areas are not

Block	ALMs	Memory Block Bits Kb	Number in system
Network Router	850	0.8	8
DDR2 Controller	4300 [1]	220.7	2
Serial Link Transceiver	4300 [1]	6.8	1

Table 6.4: Non-core block resource utilisation, numbers given are per block

not provide a definitive answer. Table 6.4 gives the area and memory utilisation for other non-core blocks in the system. In this section when discussing memory sizes a capital B refers to bytes, a lower case b refers to bits. The K prefix signifies a block of 1024 bytes or bits. All area figures are rounded to the nearest 100. The FPGA synthesis tool usually gives different areas for the same piece of HDL instantiated in different places and in different designs. Where several possible figures exist they were averaged before being rounded.

Logic utilisation is measured in adaptive logic modules (ALMs). These are the basic building block of the Stratix IV FPGA. They consist of two 6-input LUTs, two adders and two registers [4]. The LUTs can be programmed to implement combinational logic or in some ALMs the SRAMs that implement them can be directly used as memory. Both the MIPS64 and Mamba register files have their memory implemented in ALMs (so register file memory is not counted in the block memory bits in table 6.3). The Stratix IV also has separate dedicated memory blocks which are used to implement cache memories as well as most FIFOs (in general the synthesis tool makes the decision of what to use to implement a FIFO, some small FIFOs may be implemented purely in registers).

As Bluespec eradicates all hierarchy and produces a flat Verilog file it is not possible to precisely break down the resource utilisation of all parts of the system. The network routers were implemented in an entirely separate Bluespec module so resource utilisation could be tabulated for them. Compared to MIPS64 Mamba used 20% more ALMs. This is due to the logic required for the scheduler, the local requester (which routes incoming memory actions to the appropriate place inside the course), the implementation of presence bits semantics and the extra presence bit cache's control logic.

Block memory usage has been broken down into cache and non-cache memories. Both Mamba and MIPS64 contain separate 16 KB data and instruction caches. Mamba has a further 4 KB presence bit cache leading to a 12% higher usage of block memory. The given cache sizes are for the actual stored data, there is additional overhead for storing tags and valid and dirty bits.

Mamba's non-cache block memory usage is 890% higher than MIPS64. The vast majority of this is taken up by the ready queue, used to store AFs that are ready to be run (backed by DDR2 memory if it overflows). This consists of 511 64-bit words giving a total memory usage of 31.9 Kb. If seeking to reduce the memory requirements of a Mamba core this is a good target for optimisation, the queue depth could be shrunk and it does not need to be 64 bits wide. Only the local part of the AF needs storing. A 64-bit wide queue was used because it simplified the implementation.

If the ready queue is entirely removed from the non-cache memory usage of Mamba, non-cache memory usage is 26% lower than MIPS64. This lower usage is due to the different internal network request buffering used by Mamba and MIPS64. In Mamba all network requests and responses are buffered in the same FIFOs. In MIPS64 read, write and CAS requests are separated into different FIFOs. This was done because each of these requests is a different size and the different FIFOs eased implementation.

The register file of MIPS64 requires 2 read ports and 1 write port. In Mamba a register file with 3 read ports and 2 write ports is required. In the FPGA implementation this leads to a 50% higher usage of ALMs for the Mamba register file compared to the MIPS64 register file.

In summary Mamba has a reasonable overhead in terms of logical utilisation but it is not excessive and it is likely it could be reduced were an effort made to do so. The increase in terms of cache memory is also reasonable. The non-cache memory overhead is huge due to the AF ready queue, however this can be significantly reduced and non-cache memory only accounts for around 10% of total core memory usage in Mamba.

6.5.1 Timing Overheads

As with logic utilisation no particular effort was made to optimise the frequency of either Mamba or MIPS64. Parts of the Mamba system may well have longer critical paths producing a slower system. The most likely issue in the Mamba architecture is the local requester which routes memory actions in the Mamba core, though it would be possible to pipeline this if it proved necessary. However that said the FPGA synthesis reports a maximum frequency of 39.38 MHz for MIPS64 and 34.54 MHz for Mamba, in all of the benchmarks below both systems were run at 25 MHz.

6.6 The benchmarks

Four benchmarks have been used in the evaluation, three microbenchmarks based around the lock, FIFO and barrier implementations and one based on the SAT solver. For every benchmark all threads involved are created at the beginning and do not get destroyed. The thread creation time is not included in the run-time of any benchmark.

6.6.1 The lock benchmark

An implementation of a lock for Mamba using notify chains based around the MCS lock [73] is described in the software techniques chapter. An MCS lock is constructed for the MIPS64 system. In the lock benchmark a single lock is created per core involved to protect a shared area in the local memory area of that core's node. The shared area consists of a number of integers. Every thread in the benchmark chooses a shared area at random and attempts to acquire the lock. Once the lock is acquired, the thread may enter the shared area. It sums the integers within the area and writes the result of the sum to the area. It then randomly generates more integers and releases the lock exiting the area (the intention being to simulate some work being done requiring memory associated with the shared area involving a mixture of computation that can be done locally without reference to the shared area and computation that directly access the area). Once a thread has left a shared area it repeats the process again for another randomly chosen shared area up to a certain number of rounds.

The benchmark is executed with a varying number of threads (8, 16, 80 and 800) over a varying number of cores (1,2,4,8). The number of rounds each thread does varies with total thread count so the total number of rounds executed by a core remains constant. The total number of rounds is 800 so 100 rounds are done with 8 threads and 1 round is done with 800 threads. After all rounds are completed the number of integers summed and generated within the shared area is increased and the rounds are repeated. The number of integers starts at 10 and increases in increment of 10 to 500. For each thread and core number the total run-time was measured as a number of cycles for each different number of integers.

In the MIPS64 implementation of the benchmark as the scheduler has no information on which thread will gain the lock next and thus should be scheduled a situation where most or all threads currently scheduled are just spinning on the lock could occur. They may also impede the progress of any other threads by generating many, effectively useless, memory operations. To alleviate this a small delay was introduced into the spin loop and a count is kept of how many times the thread has spun. Once it has spun 100 times the thread yields to the scheduler, scheduling another thread in its place. The number of spins until yield was experimentally determined to be the one that gave the best results.

6.6.2 The barrier benchmark

The barrier benchmark tests the performance of the barrier implementation described in the software techniques chapter. The MIPS64 implementation is similar to the Mamba implementation. To wait on the barrier in the MIPS64 implementation a thread first acquires a spin lock. Once the lock is acquired a count is increased, if this isn't yet the total expected the thread attempts to acquire an MCS lock (releasing the spin lock before waiting on it). This lock has been setup so it starts locked meaning the thread will immediately begin to wait on it. When a thread increases the count of threads waiting on the barrier to the total expected it unlocks the MCS lock. Every other thread waiting on the barrier is attempting to acquire this lock and when they succeed they immediately release it. This has the effect of every thread waiting on the MCS lock (and thus the barrier) being woken. The overall situation is very similar to the Mamba barrier which uses a notify chain with the notable difference that every MIPS64 thread is actively spinning on a location whilst the Mamba threads sleep.

The barrier consists of a number of threads waiting on a single barrier over a number of rounds. The time each thread takes to do all of the rounds is measured. Measurement starts from the second round, this is because the time of the first round will include the setup time to create all of the threads and will be longer than usual. The benchmark is executed with a varying number of cores (1-8) with varying numbers of threads (1, 2, 3, 4, 8, 16, 32, 64).

As with the lock benchmark the MIPS64 barrier will suffer when the scheduler schedules only threads that are waiting on the MCS lock and the next thread to be woken has not been scheduled. So a delay is introduced into the spin loop and a yield to the scheduler after a certain number of spins exactly as it was for the lock benchmark.

6.6.3 The FIFO benchmark

The FIFO benchmark tests the latency and throughput of the FIFO described in the software techniques chapter. The MIPS64 implementation of the FIFO is that of Lamport [58]. This is the FIFO design that formed the basis of the Mamba implementation. The benchmark designates a single core as the consumer core and a varying number of cores as the producer cores. For each producer core a certain number of threads are created and a matching thread created on the consumer core (so if there are c producer cores and t threads per producer core there will be $c \quad t$ threads created on the consumer core). Each producer thread sends a number of integers (100'000 in the benchmarks presented here), for the throughput measurement the consumer threads receive and discard the integer, for the latency measurements the consumer threads send the integer back to the producer thread that sent it (two separate FIFOs are used one for each direction). When testing latency a producer thread will wait until it has received its integer back before sending the next.

The benchmark is executed with a varying number of producer cores (1-7) with varying numbers of threads per producer core (1, 2, 3, 4, 8, 16, 32), there is always a single consumer core. The time each thread takes to execute is measured separately, and the total number of cache reads and writes are measured for each core.

The MIPS64 implementation of the benchmark does not include a scheduler yield or a delay in the loop where a producer or consumer is waiting for the FIFO to become non-empty or non-full. This wasn't done because doing so reduced performance (for the throughput benchmark this indicates a producer or consumer rarely needed to wait for the FIFO to become non-empty or non-full and for the latency benchmark this indicates it effected the responsiveness as a thread was more likely to be descheduled before it could reply back)

It is important to realise that the microbenchmarks are not intended to show a decreasing run-time with increasing thread count. For the FIFO benchmarks we are measuring throughput and latency, raw runtime actually increases with increasing thread count and core count because each thread sends 100000 words each. So each extra thread and core increases the total amount of work done. For the lock and barrier benchmarks we are interested in the system's sensitivity to thread count. In the lock benchmark as the lock is a serialising operation simply adding more threads wont decrease run-time. For the barrier benchmark because it performs a linear wake up of threads wed expect a linear increase in run-time with thread count.

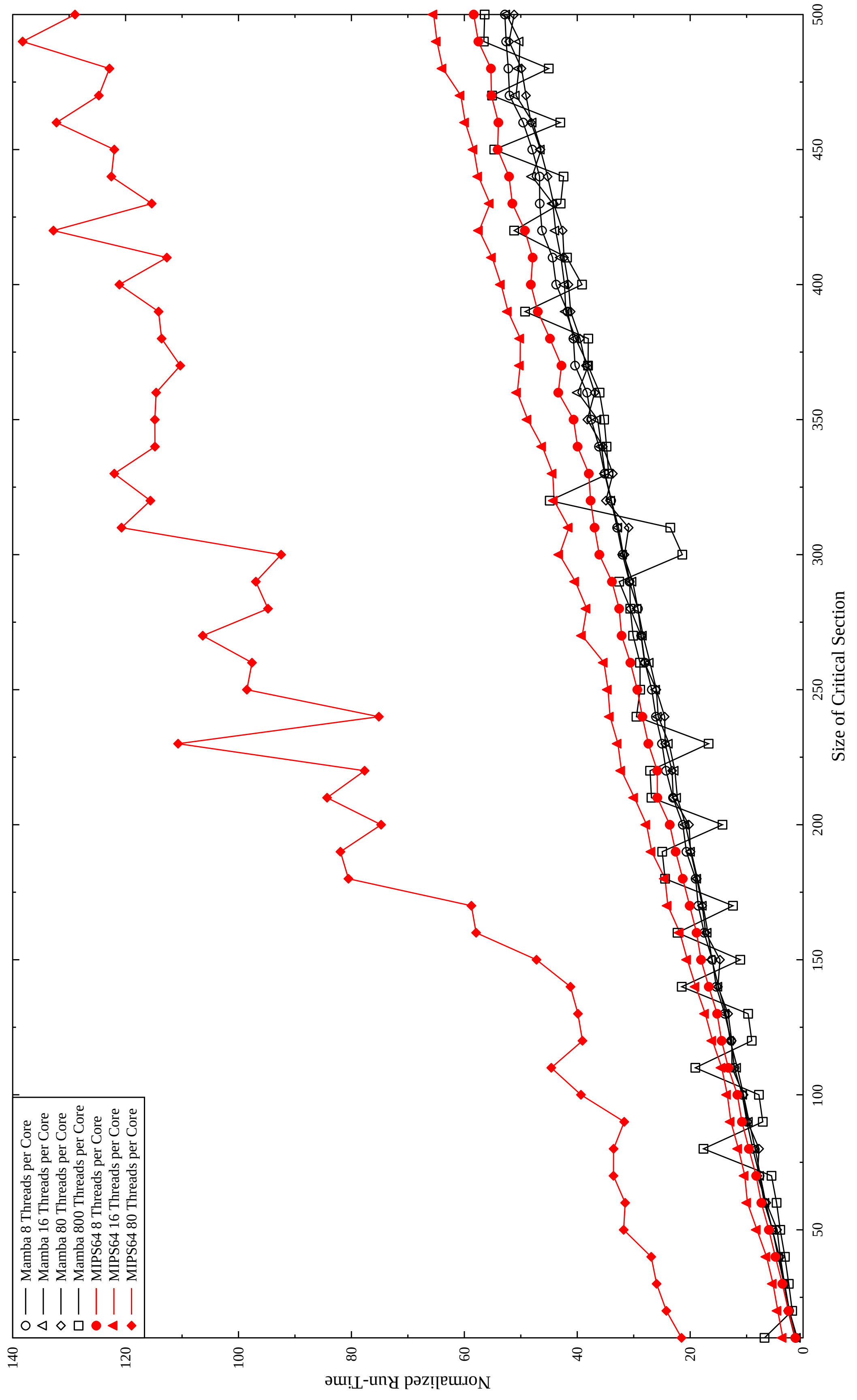


Figure 6.3: Run-time of the lock benchmark with increasing critical section size running on 8 cores. Run-time is normalised to MIPS64 run-time on 1 core with smallest critical section size with 8 threads

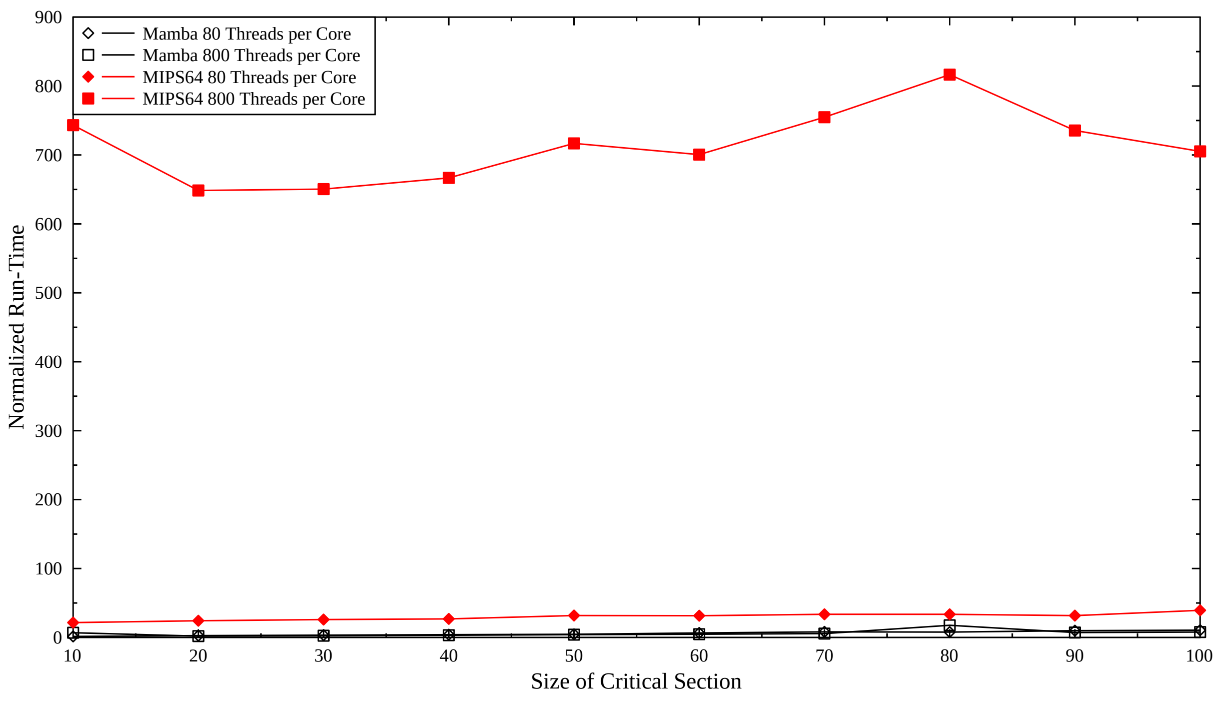


Figure 6.4: Run-time of the lock benchmark with increasing critical section size running on 8 cores for 80 and 800 threads. Run-time is normalised to MIPS64 run-time on 1 core with smallest critical section size with 8 threads

6.7 Results

6.7.1 The lock benchmark

Results of running the lock benchmark over 8 cores can be seen for 8, 16 and 80 threads for Mamba and MIPS64 in figure 6.3. The size of critical section refers to the number of integers summed and generated within the shared area when a thread acquires that shared area's lock. The results show that for Mamba thread count does not have much of an effect on performance. Whilst there is more variation in run-time for 800 threads it still performs at the same level as the 8 thread run. For MIPS64 the difference in performance between the 16 threads and 80 threads is huge. For 8 threads the MIPS64 version achieves similar performance to the Mamba version, with MIPS64 between 5% and 13% slower, at 16 threads the MIPS64 version is between 19% and 250% slower. Though the biggest difference occurs at the smallest critical section size. As the size of the critical section increases Mamba the MIPS64 get closer in performance for 16 threads. For 80 threads the MIPS64 implementation is between 150% and 1390% slower, with the biggest difference occurring at a critical section size of 10 and the smallest difference occurring at a critical section size of 480.

Result for 800 threads in MIPS64 have been left off the graph, this is because they cannot sensibly fit. A graph comparing Mamba and MIPS64 for 80 and 800 threads only over 8 cores can see in figure 6.4. Here the run-time is only measured up to a critical section size of 100 because the run-time was getting large and the trend is clear so there is no need

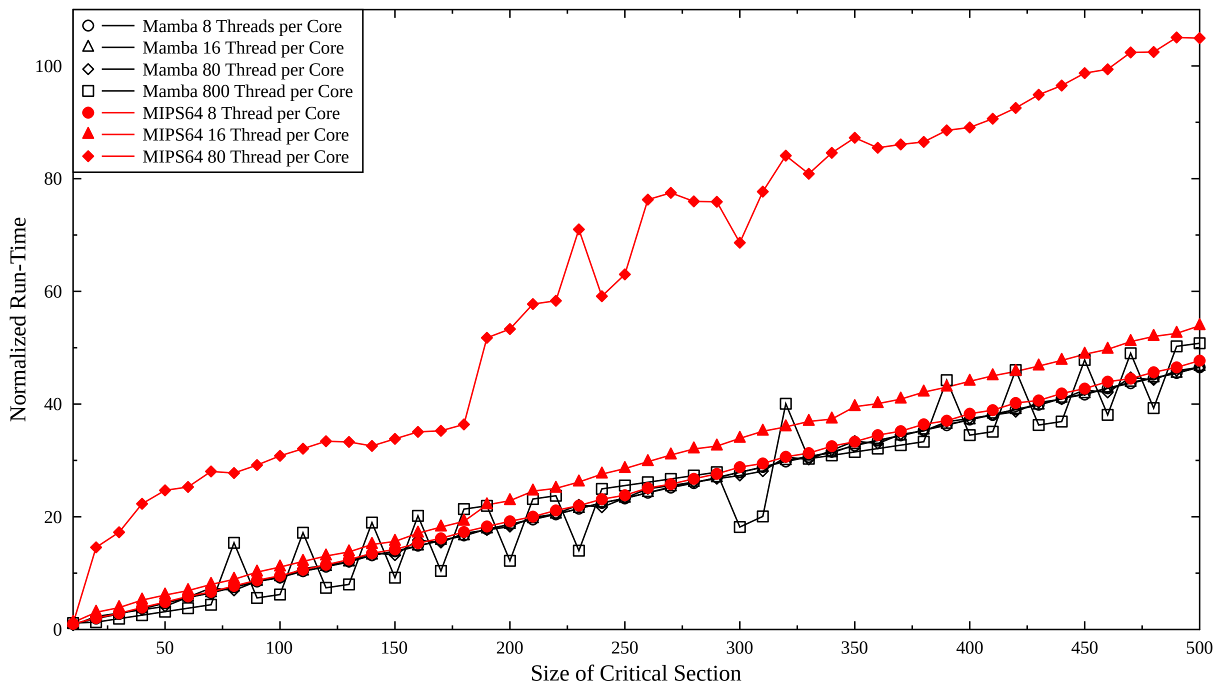
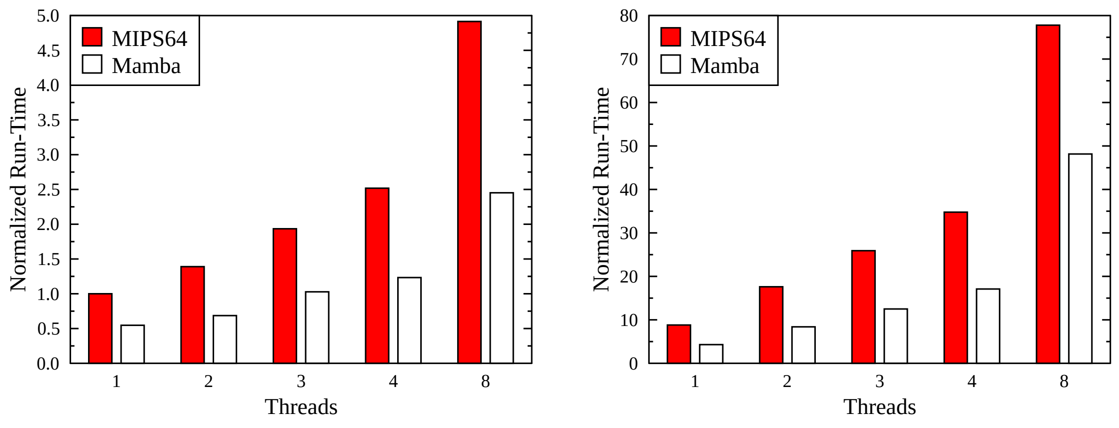


Figure 6.5: Run-time of the lock benchmark with increasing critical section size running on 1 core. Run-time is normalised to MIPS64 run-time on 1 core with smallest critical section size with 8 threads

to take it to 500. At its fastest MIPS64 is 46 slower than Mamba (critical section size of 60) and at its slowest MIPS64 is 256 slower than Mamba (critical section size of 40).

There are various critical section sizes where Mamba performs best at 800 threads. This is because at 800 threads each thread only attempts to acquire a single lock once. This means all of the lock contention happens at the beginning of the benchmark when all threads attempt to acquire one of the available locks. Once the notify chain is setup no further contention occurs. If the shared areas randomly chosen by each thread happen to be in that thread's local memory or in a close neighbour than the single run through the shared area will be fast, it is these cases than show the best performance. When the number of rounds a thread does increases as the total thread count is decreased lock contention will happen throughout the benchmark and it is less likely that a single thread will always access shared areas that are in its local memory or nearby, so the performance trend is more uniform.

The lock benchmark doesn't show much sensitivity to core count in terms of how the benchmark behaves with increasing thread number and critical section size at a given core count. The results for 1 core can be see in figure 6.5, again at 8 and 16 threads there is not much difference in performance between Mamba and MIPS64. For 8 threads MIPS64 is between 6% and 2% slower and for 16 threads MIPS64 is between 13% and 47% slower (again the biggest difference occurring at the smallest critical section size, the gap quickly closes as the size of the critical section is increased). For 80 threads MIPS64 is between 36% and 740% slower (biggest difference at critical section size of 20, smallest at 10). At 800 threads MIPS64 is between 2 (critical section size of 10) and 143 (critical



(a) Run-time with 1 core (b) Run-time with 8 cores

Figure 6.6: Barrier benchmark normalised run-time for up to 8 threads for 1 and 8 cores. Run-time normalised to MIPS64 1 thread run-time

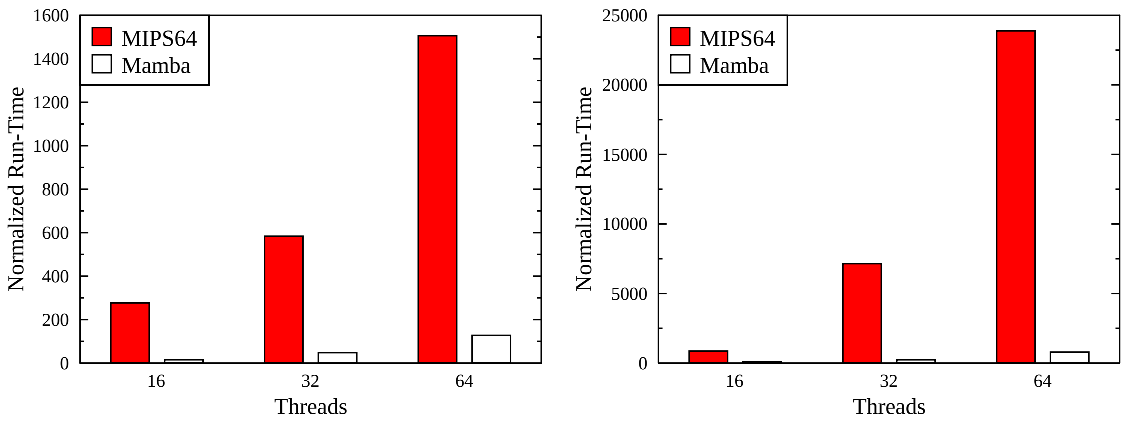
section size of 70) slower than Mamba.

The performance between numbers of cores whilst keeping the thread number constant can also be compared, with performance differences between 1% and 25% for Mamba and 6% and 44% for MIPS64, except at small critical section sizes where sensitivity to core count can be seen. In MIPS64 beyond 8 threads with a critical section size of 10 comparing 8 core run-time vs. 1 core run-time there is a difference of 187% at 16 threads, a difference of 20 at 80 threads and a difference of 258 at 800 threads. In Mamba at 800 threads 8 cores is 700% slower than 1 core (at a critical section size of 10) but this gap rapidly closes to the 1% to 25% range otherwise performance difference between cores stays in the 1% to 25% range.

Originally the MIPS64 version of this benchmark performed worse. Each thread created a node on that stack that that thread spins on whilst waiting for the lock (the MCS lock is a list of these nodes waiting for the lock). Due to the way memory is allocated the address of each thread's node aliased to the same line in the cache (as it is direct mapped), so whilst several threads were spinning on their nodes performance was degraded due to the continuous cache misses. This problem was corrected (nodes are placed elsewhere in a manner that does cause them all to be in the same cache line) to give the results presented here. The Mamba implementation also creates the nodes of the notify chain on the stack, and these will also all be placed in the same cache line, however it is not an issue within Mamba as the threads do not actively spin on the node.

6.7.2 The barrier benchmark

The performance difference in the barrier benchmark between Mamba and MIPS64 is similar for up to 8 threads. The trend in performance with increasing thread number is similar for the different numbers of core. Graphs of the normalised run-time of the benchmark for 1 and 8 cores for up to 8 threads can be seen in figure 6.6. For up to 8 threads between 1 - 8 cores MIPS64 is between 260% and 320% slower than Mamba.



(a) Run-time with 1 core

(b) Run-time with 8 cores

Figure 6.7: Barrier benchmark normalised run-time for 16, 32 and 64 threads for 1 and 8 cores. Run-time normalised to MIPS64 1 thread run-time

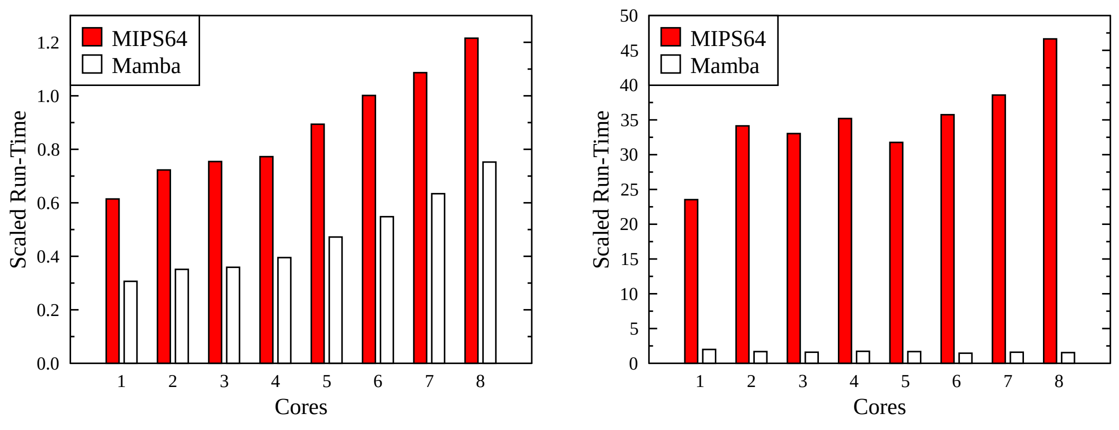
Beyond 8 threads the performance of MIPS64 rapidly becomes worse, figure 6.7 shows the run-time for 1 and 8 cores for 16, 32 and 64 threads. With 1 core at 16 threads MIPS64 is 17 slower and with 8 cores at 64 threads MIPS64 is 46 slower. In figure 6.8 the normalised run-time is divided by the total number of threads (number of cores number of threads) and graphed for 8 thread and 64 threads with increasing core count. This demonstrates how well the barrier implementation scales. At 8 threads both MIPS64 and Mamba scale well with no drastic differences in scaled run-time. At 64 threads per core Mamba continues to exhibit good scaling whilst MIPS64 does not.

The likely cause of MIPS64's poor performance in this benchmark is the linear nature of the barrier wake up operation. If the next thread to be woken is not scheduled then time is wasted whilst other threads spin. A more efficient barrier would involve a tree structure, where each thread wakes up two or more threads. Still at 8 threads and below where no scheduling occurs Mamba can still outperform MIPS64 by up to 100%. A tree implementation of the barrier would greatly improve MIPS64's performance at higher thread counts but the tree implementation would also benefit Mamba.

6.7.3 The FIFO benchmark

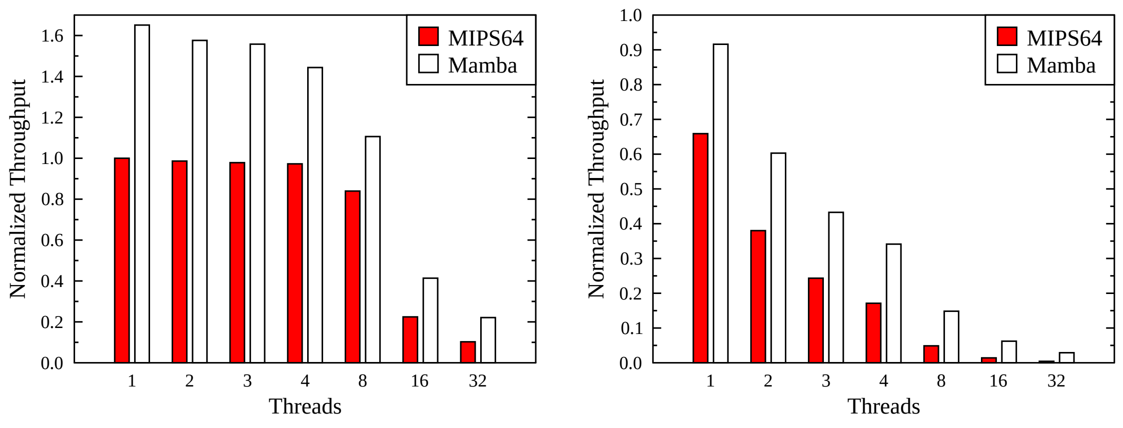
Throughput

The run-time of every thread in the FIFO benchmark is measured and averaged with each producer thread sending a continuous stream of integers and each consumer thread consuming an integer and immediately discarding it. Throughput is then the reciprocal of the run-time. The measured throughput for increasing threads per producer core can be seen for 1 producer core and 7 producer cores in figure 6.9. For 1 producer core MIPS64 has between 115% and 31% worse throughput than the Mamba implementation, with the smallest difference occurring at 8 threads per producer core and the largest at 32 threads per producer core. For 7 producer cores MIPS64 has between 39% and 770%



(a) Scaled run-time for 8 threads per core (b) Scaled run-time for 64 threads per core

Figure 6.8: Barrier benchmark scaled run-time for 1 - 8 cores with 8 and 64 threads. Normalised run-time is divided by the total number of threads to get scaled run-time



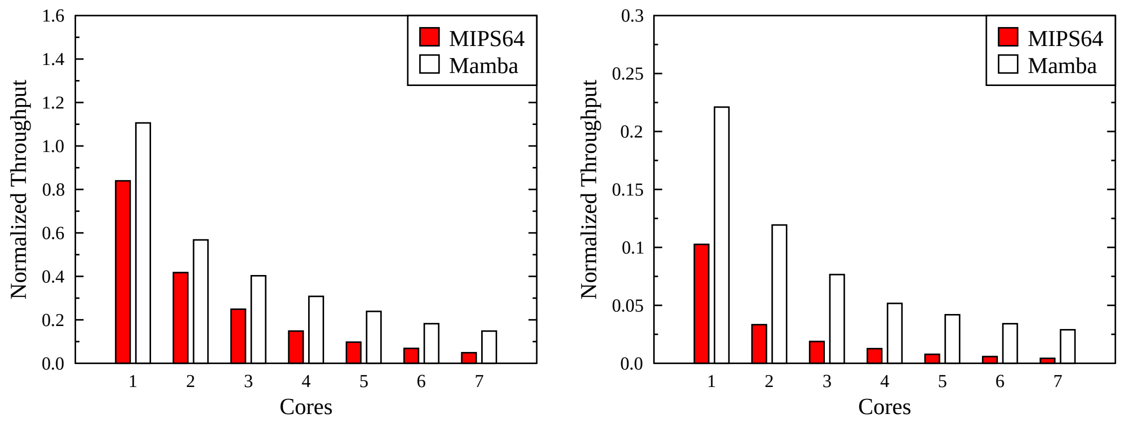
(a) Throughput with 1 producer core (b) Throughput with 7 producer cores

Figure 6.9: Throughput of FIFO benchmark for 1 and 7 producer cores for 1, 2, 3, 4, 8, 16 and 32 threads per producer core. Throughput is normalised to MIPS64 run with 1 producer core with 1 thread per producer core

worse throughput with the smallest difference occurring at 1 thread per producer core and the largest at 32 threads per producer core.

In figure 6.10 thread count per producer core is held constant (at 8 and 32 threads per producer core), whilst the number of producer cores varies. For 8 threads per producer core MIPS64 has between 31% and 204% worse throughput than Mamba, with the smallest difference occurring at 1 producer core and the largest at 7 producer cores. For 32 threads per producer core MIPS64 has between 115% and 770% worse throughput than Mamba with the smallest difference occurring at 1 producer core and the largest at 7 producer cores.

It is notable that throughput is significantly worse for both Mamba and MIPS64 when comparing equal thread counts on differing numbers of producer cores. This is to be expected as as the number of producer cores increases the number of consumer threads



(a) Throughput with 8 threads per producer core(b) Throughput with 32 threads per producer core

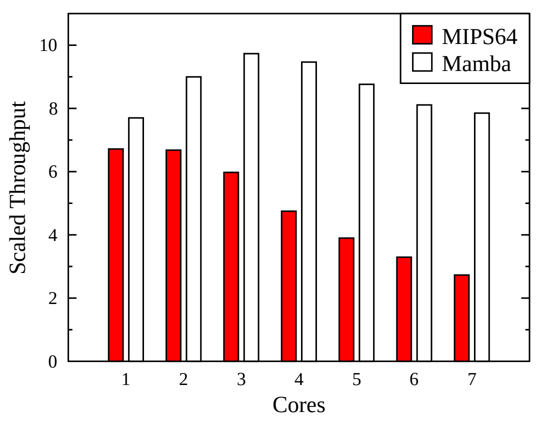
Figure 6.10: Throughput of FIFO benchmark 8 and 32 threads per producer core for 1 - 7 producer cores. Throughput is normalised to MIPS64 run with 1 producer core with 1 thread per producer core

(which are all the same core) also increases. The bottleneck formed by the single consumer core is the reason that throughput gets worse as the number of producer cores is increased. In figure 6.11 the throughput is scaled by the total number of producer threads (producer threads per core producer cores) to give a measure of the total throughput achieved by all FIFOs rather than the throughput of an individual FIFO. This demonstrates that the Mamba implementation exhibits significantly greater resilience to the total increase in producer threads. For Mamba in both the 8 threads per producer core and 32 threads per producer core cases the scaled throughput is slightly lower with 7 producer cores compared to 1 producer core with a peak in the middle. The difference between the highest and lowest scaled throughput is 18% for 8 threads per producer core and 17% for 32 threads per producer core. For MIPS64 in both the 8 threads per producer core and 32 thread per producer core cases the scaled throughput drops off immediately. The difference between the highest and lowest scaled throughput is 350% for 8 threads per producer core and 440% for 32 threads per producer core.

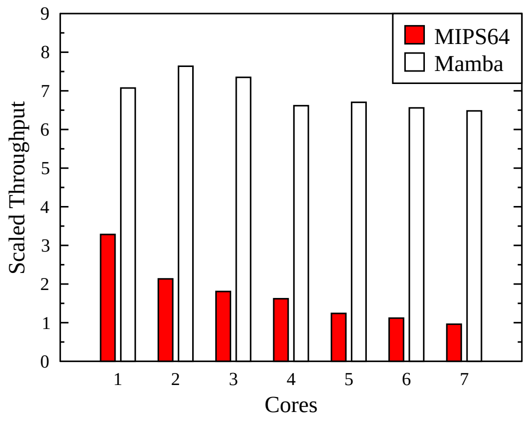
Overall Mamba is maintaining roughly the same total throughput. As the number of producer threads increases throughput per thread goes down but in proportion to the increase in producer threads. The same is not true of MIPS64.

Latency

For the latency measurement every time a consumer thread consumes an integer it immediately sends it back via another FIFO. The producer thread waits for the response from the consumer thread before producing the next integer. The run-time of all threads is measured and averaged, this is directly proportional to the average latency. The measured latency for increasing threads per producer core can be seen for 1 producer and 7 producer cores in figure 6.12. For 1 producer core for up to 16 producing threads MIPS64 has between 81% and 320% higher latency than Mamba, the smallest difference occurs at 8 threads per producer core and the largest at 16 threads per producer core. At 32

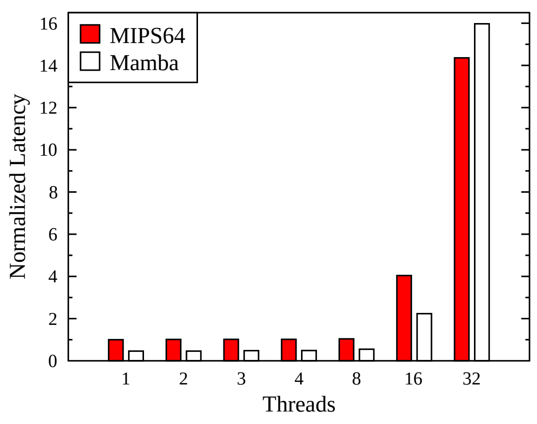


(a) Scaled throughput with 8 threads per producer core

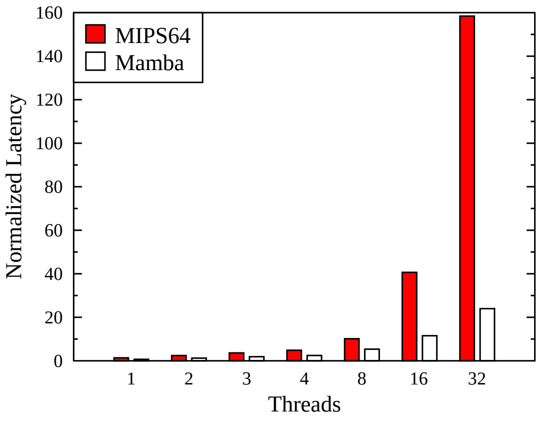


(b) Scaled throughput with 32 threads per producer core

Figure 6.11: Throughput of FIFO benchmark scaled by the total number of producer threads with 8 and 32 threads per producer core for 1 - 7 producer cores. Throughput is normalised to MIPS64 run with 1 producer core with 1 thread per producer core



(a) Latency with 1 producer core

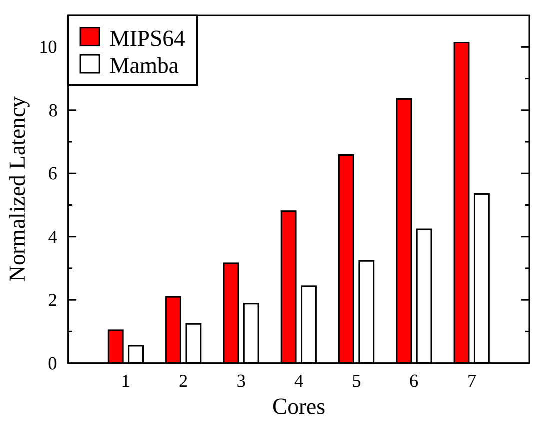


(b) Latency with 7 producer cores

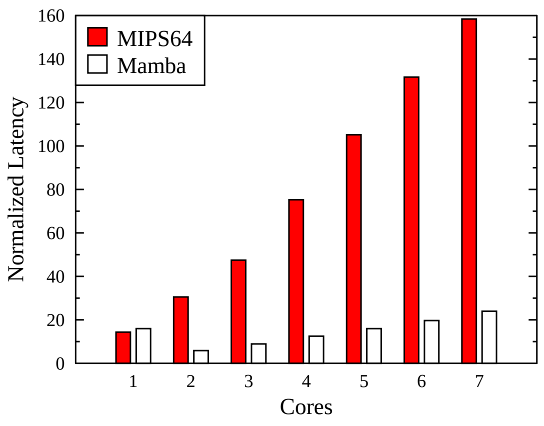
Figure 6.12: Latency of FIFO benchmark for 1 and 7 producer cores for 1, 2, 3, 4, 8, 16 and 32 threads per producer core. Latency is normalised to MIPS64 run with 1 producer core with 1 thread per producer core

producer threads Mamba performs slightly worse than MIPS64 with 11% greater latency. For 7 producer cores MIPS64 has between 94% and 760% higher latency than Mamba, the smallest difference occurs at 1 thread per producer core and the largest at 32 threads per producer core.

Figure 6.13 shows how latency changes with increasing producer core count with the number of threads per producer core held constant. For 8 thread per producer core MIPS64 has between 68% and 300% worse latency with the smallest difference occurring at 3 producer cores and the largest different occurring at 5 producer cores. For 32 threads per producer core MIPS64 has between 620% and 770% worse latency apart from at 1 producer core where MIPS64 is 11% better, otherwise the smallest difference occurs at 2



(a) Latency with 8 threads per producer core



(b) Latency with 32 threads per producer core

Figure 6.13: Latency of FIFO benchmark 8 and 32 threads per producer core for 1 - 7 producer cores. Latency is normalised to MIPS64 run with 1 producer core with 1 thread per producer core

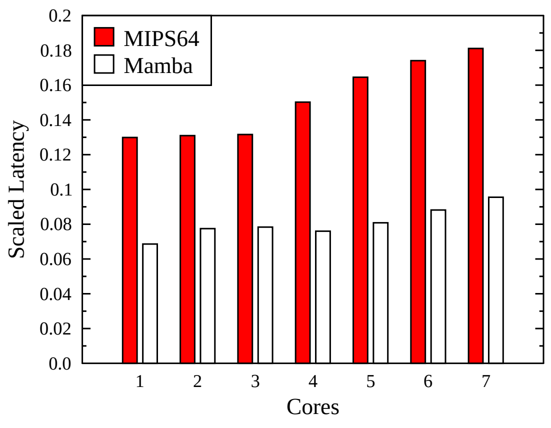
producer core and the largest at 6 producer cores.

As with throughput a scaled result can be used to determine how well the latency of the FIFO implementation scales with increasing total thread count. The normalised latency is divided by the total number of producer threads (producer threads per core producer cores) to give scaled figure. The result of this can be seen in figure 6.14 for 8 threads and 32 threads per producer core. Both Mamba and MIPS64 exhibit reasonable scaling here. For 8 threads per producer core there is a maximum of 39% difference in scaled latency for MIPS64 (difference between 1 producer producer core and 7 producer cores) and a maximum of 39% difference in scaled latency for Mamba (difference between 1 producer core and 7 producer cores). For 32 threads per producer core there is a maximum of 57% difference in scaled latency for MIPS64 (difference between 1 producer core and 7 producer cores) and a maximum of 20% difference for Mamba (difference between 2 producer cores and 7 producer cores) ignoring the 1 producer core case (which is 78% worse than the best). Though in the 32 threads per producer core case there is a noticeable upward trend in scaled latency for MIPS64, the trend for Mamba appears flatter.

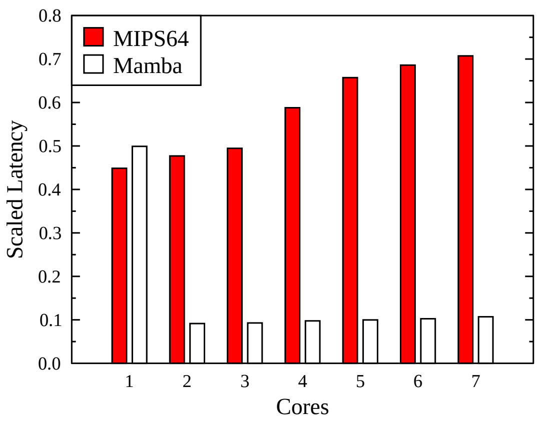
Memory Operations

Also of interest in the FIFO latency and throughput benchmarks are the memory operations, the number of reads and writes that occur in each cache. For the Mamba FIFO produce and consume operations a memory access should only occur when an item can be placed or removed from the FIFO. If this is not possible the producer or consumer thread using the FIFO should stall until it is able to continue. So producing or consuming an item will use a constant amount of memory accesses regardless of how long it takes. Unlike in the polling version where the longer a thread waits to produce or consume the more memory operations are used.

For the throughput benchmark all the FIFOs are contained in the address space of the consumer core. Figure 6.15 shows the measured memory operations in the data cache of

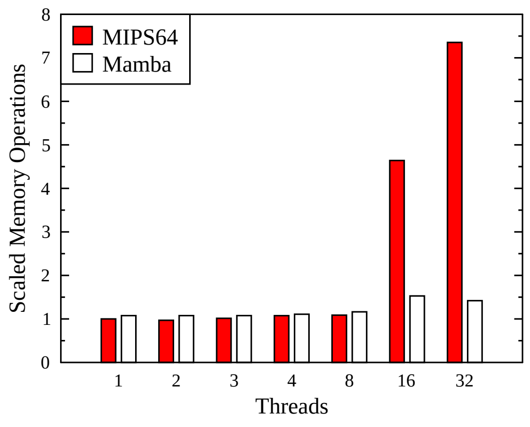


(a) Scaled latency with 8 producer threads per core

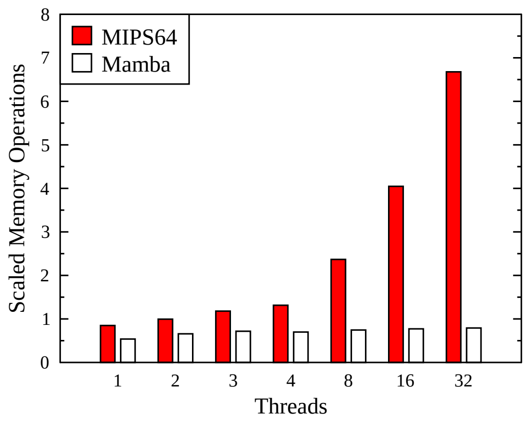


(b) Scaled latency with 32 producer threads per core

Figure 6.14: Latency of FIFO benchmark scaled by dividing by the total number of producer threads with 8 and 32 threads per producer core for 1 - 7 producer cores. Latency is normalised to MIPS64 run with 1 producer core with 1 thread per producer core



(a) Scaled memory operations with 1 producer core

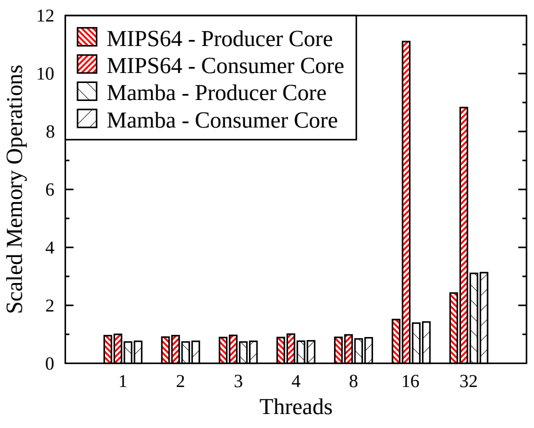


(b) Scaled memory operations with 7 producer cores

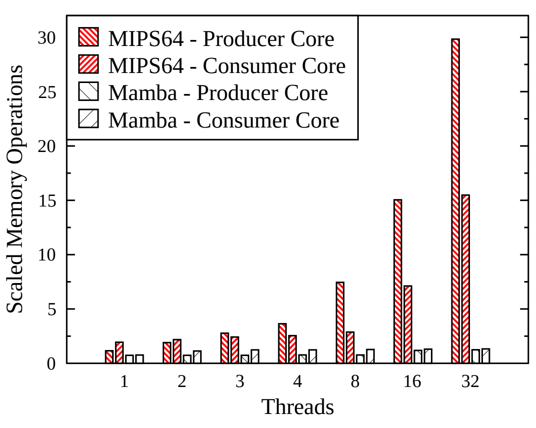
Figure 6.15: Memory operations in the data cache of the consumer core divided by the total number of producer threads (total number of FIFOs) in the throughput benchmark. Normalized to the MIPS64 consumer core on the 1 producer core and 1 producer thread run

the consumer core for 1 and 7 producer cores divided by the total number of producer threads (threads per producer core producer cores) which is the total number of FIFOs.

The scaled memory operations remain similar for all numbers of threads for Mamba, this is not the case for MIPS64. For 1 producer core the biggest difference in scaled memory operations for Mamba is 42%, it is 860% for MIPS64. At 7 producer cores Mamba has a maximum difference of 47% in scaled memory operations and MIPS64 has a maximum difference of 890%.



(a) Scaled memory operations with 1 producer core



(b) Scaled memory operations with 7 producer cores

Figure 6.16: Memory operations in the data cache of the consumer core and average producer core divided by the total number of FIFOs in the core's address space in the latency benchmark. Normalized to the MIPS64 consumer core on the 1 producer core and 1 producer thread run

Mamba initially performs slightly more operations than MIPS64, however at higher thread counts MIPS64 uses significantly more. For 1 producer core at 1 thread per producer core Mamba uses 8% more operations but at 32 threads per producer core MIPS64 uses 620% more. For 7 producer cores at 1 thread per producer core MIPS64 uses 58% more memory operations than Mamba and at 32 threads per producer core MIPS64 uses 940% more. It is also notable that Mamba uses fewer operations at 7 producer cores compared to 1 producer core, with a difference of 99% at 1 thread per producer core and a difference of 79% at 32 threads per producer core (The two graphs have the same scale so can be compared directly).

For the latency benchmark again all of the FIFOs written to by the producer threads are in the address space of the consumer core but the other FIFOs written to by the consumer threads in response to the producer threads are placed in the address space of the corresponding producer cores. The scaled memory operations from the data cache of the consumer core and the average of the producer cores are presented here in figure 6.16 for 1 and 7 producer cores.

In Mamba there is not much difference in scaled memory operations between the consumer and producer cores compared to MIPS64. The greatest difference for 1 producer core is 5% at 8 threads per producer core and 68% for 7 producer cores at 3 thread per producer core. For MIPS64 the greatest difference is 840% at 1 producer core with 16 threads, and 360% for 7 producer cores at 8 threads per producer core.

For 1 producer core at 8 threads and below Mamba and MIPS64 have similar amounts of scaled memory operations, the biggest difference occurs at 16 threads per producer core where the MIPS64 consumer core has 880% more scaled memory operations than the Mamba consumer core. At 7 producer cores Mamba mostly has significantly less scaled memory operations with the smallest difference being 56% between the average MIPS64 and Mamba producer core at 1 thread per producer core and the largest difference being

23.9 between the average MIPS64 and Mamba producer core at 32 threads per producer core.

Again the scaled memory operations remain fairly similar for Mamba regardless of thread or core number apart from at 1 producer core with 32 producer threads. At 1 producer core between 1 and 16 producer threads the maximum difference in memory operations is 88%, for 32 producer threads there 520% more memory operations than the lowest amount (at 1 producer thread). At 7 producer cores the biggest difference seen is 73%. This offers a possible explanation for the poorer performance of the Mamba implementation at 32 producer threads with 1 producer core. Excess context swapping will generate more memory operations so it may be that that particular setup causes bad scheduling behaviour where threads get swapped whilst still running (i.e. not stalled waiting for something to become present) so there is always something in the ready queue so context swaps happen more often.

For MIPS64 scaled memory operations do not remain the same for differing thread and core numbers. For 1 producer core the biggest difference seen is 11.6 and for 7 producer cores the biggest difference seen is 25.8 .

6.7.4 Microbenchmark summary

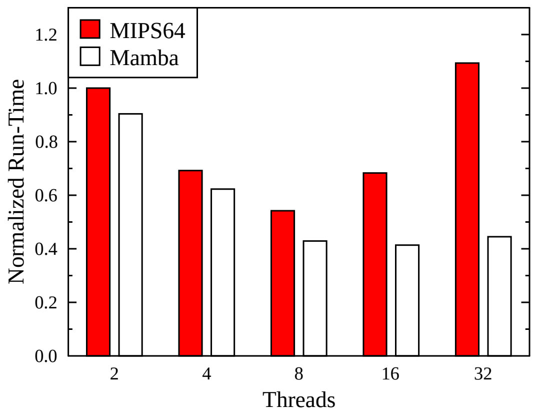
From the lock and barrier benchmarks it is clear than beyond 8 threads when the software scheduler of MIPS64 will start performing context switching in software, performance rapidly diverges between Mamba and MIPS64. This is because the Mamba scheduler, informed by presence bits, makes more intelligent decisions about when to switch in a thread. Still even with 8 threads per core or below Mamba can show superior performance, as seen in the FIFO latency and throughput benchmarks at the lower thread and core numbers. Again the difference in Mamba and MIPS64 becomes significantly larger as the number of threads in the FIFO benchmarks is increased.

The ability for Mamba to stall a thread waiting for a word to become present allows it to keep the memory operations required roughly proportional to the total amount of work done for FIFO communication due to the lack of polling. This is part of the reason Mamba shows superior performance to MIPS64. The ability for presence bits to give information to the scheduler about what threads are ready to run also helps increase responsiveness as seen in the latency benchmark.

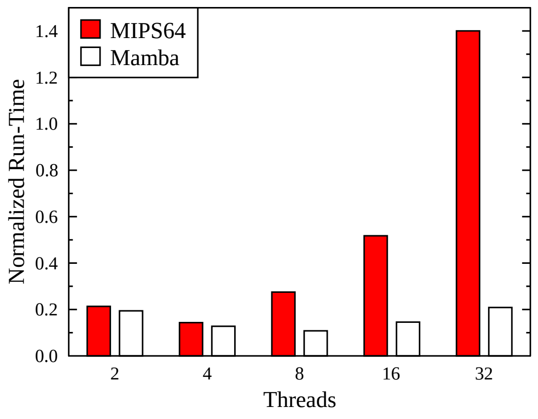
Mamba also scales well with increasing thread count. In the lock benchmark work being done remains constant, it is just split over more threads and so the run-time of the benchmark remains roughly similar with increasing thread count. With the FIFO benchmarks work done increases with thread count (as each thread sends the same amount of numbers through a FIFO) but when scaled by the total number of FIFOs being used performance remains roughly the same. This demonstrates that the number of threads spawned to do a particular amount of work is not of critical importance in Mamba.

Instance	Number of Variables	Number of Clauses
Medium	116	953
Huge	459	7054

Table 6.5: SAT blocks word instance statistics



(a) Run-time with 1 core



(b) Run-time with 8 cores

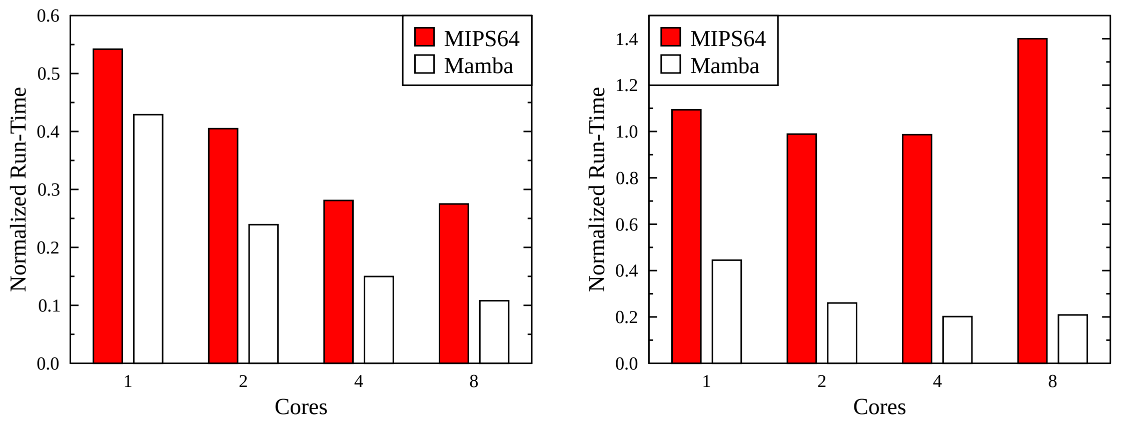
Figure 6.17: Normalised run-time of the SAT solver for 1 and 8 cores solving the 'huge' instance. Run-time normalised to the MIPS64 run with 2 threads and 1 core

6.7.5 The SAT solver

The SAT solver was run on two different problem instances taken from SATLIB [43], both are blocks world problems. A blocks world problem consists of an initial configuration of blocks and a goal configuration, a sequence of moves satisfying certain rules (can't move a block with another block on top of it, a block must be placed on top of another block etc.) that transforms the initial configuration to the goal configuration is a solution to the problem. These problems can be encoded into CNF formulae which are satisfiable if and only if there exists a solution.

The two instances taken from SATLIB are described as 'medium' and 'huge', both are satisfiable. Table 6.5 lists the numbers of variables and clauses in both instances. These instances were chosen because the huge instance takes a reasonable amount of time to solve (around 45 minutes for the slowest run) and the medium instance is sufficiently large that at the largest thread and core count examined (8 cores with 32 threads per core) each thread has some work to do, but not a large amount (there are 3.7 clauses per thread at that point) which allows an examination of performance at an extreme.

The SAT solver was run using 1, 2, 4 and 8 cores with 2, 4, 8, 16 and 32 process threads per core. Figure 6.17 shows the normalized run-time of solver for the 'huge' instance for 1 and 8 cores. In all cases Mamba performs better than MIPS64, the smallest performance difference of 10% occurring at 2 threads with 1 core and the biggest performance difference of 770% occurring at 32 threads per core with 8 cores. With 1 core Mamba and MIPS64 scale very similarly down to 8 threads per core at which point the performance of Mamba stays level but the performance of MIPS64 gets worse. At 32 threads per core Mamba



(a) Run-time with 8 threads per core (b) Run-time with 32 threads per core

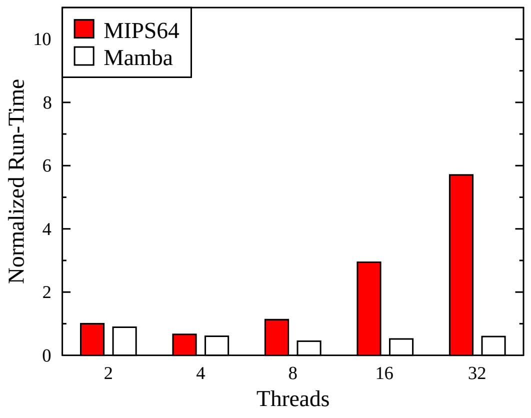
Figure 6.18: Normalised run-time of the SAT solver for 8 and 32 threads per core solving the 'huge' instance. Run-time normalised to the MIPS64 run with 2 threads and 1 core

performs 300% better compared to 1 thread per core and performs 350% better than MIPS64 at 32 threads per core, the best performance for Mamba occurs at 8 threads per core and performance at 32 threads per core is only 3% worse. MIPS64 at 32 threads per core performs 9% worse than MIPS64 at 1 thread per core with the best performance occurring at 8 threads per core, 32 threads per core performs 350% worse than 8 thread per core. So with 1 core Mamba exhibits good scaling, once it reaches the maximum performance allowable by the algorithm employed increasing the number of threads does not have much impact, for MIPS64 increasing the number of threads beyond the optimal has far greater impact.

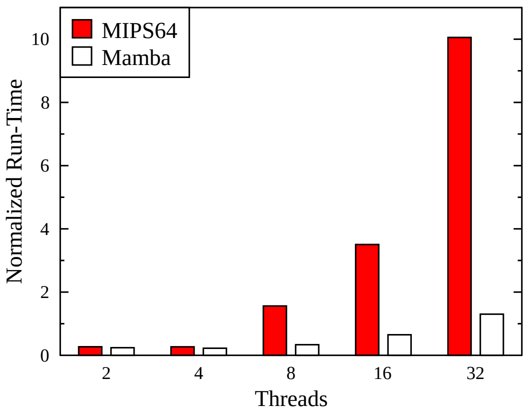
With 8 cores both MIPS64 and Mamba exhibit worse scaling. For MIPS64 the optimal run-time occurs at 4 threads per core, the worst run-time occurs at 32 threads per core and is 9.8 worse than 4 threads per core. For Mamba the best run-time occurs at 8 threads per core, 32 threads per core has the worst run-time and is 93% slower than the run-time at 8 threads per core. Still compared to MIPS64 for the same core number the scaling trend is significantly better.

Figure 6.18 shows the run-time of the solver for the 'huge' instance for 8 and 32 threads per core with increasing core number. At 8 threads per core both MIPS64 and Mamba exhibit good scaling behaviour, whilst MIPS64 is slower than Mamba performance continues to increase with increasing core count. It reaches a plateau at 4 cores though 8 cores are slightly faster, there is a difference of 97% between 1 core and 8 cores. For Mamba performance continues to get better all the way to 8 cores, with a difference of 500% between 1 and 8 cores.

At 32 threads per core MIPS64 scales poorly with increasing core count, it reaches a plateau at 2 and 4 cores but performance drops off again at 8 cores, performance at 8 cores is 42% worse than the performance at 2 and 4 cores and 28% worse than the performance at 1 core. Mamba reaches a plateau at 4 cores, performance at 8 cores is only 4% worse than performance at 4 cores and is 310% better than performance at 1 core.



(a) Run-time with 1 core



(b) Run-time with 8 cores

Figure 6.19: Normalised run-time of the SAT solver for 1 and 8 cores solving the 'medium' instance. Run-time normalised to the MIPS64 run with 2 threads and 1 core

Figure 6.19 shows the normalized run-time of the solver for the 'medium' instance for 1 and 8 cores (the graph scales are the same so the bars can be directly compared between 1 and 8 cores). The medium instance is solved far quicker than the huge instance (around 4 seconds for the slowest run compared to 45 minutes for the 'huge' instance). The SAT solver scales poorly on both MIPS64 and Mamba for the medium instance. At 1 core MIPS64 is 960% slower at 32 threads compared its best performance at 4 threads. Mamba scales better than this but still has an upward trend by 32 threads. At 32 threads Mamba is 32% slower than its best performance at 8 threads. At 8 cores MIPS64 has a sharp upward trend with its worst performance at 32 threads 37 worse than its best performance at 2 threads. Mamba hits a performance plateau at 4 threads before trending upwards with its worst performance at 32 threads 680% worse than its best performance at 4 threads.

Figure 6.20 more clearly illustrates the performance increase gained with increasing thread count in the SAT solver benchmark with the large instance. Here thread count is the total number of threads in the system, i.e. threads per core cores. It starts at 2 threads on 1 core. For each further point the number of threads and cores doubles til it reaches 16 threads on 8 cores. The final point of 256 threads is 32 threads on 8 cores. Whilst it may seem that Mamba only offers modest gains in performance given the greatly increased thread count it is important to realise than at 5 threads or more per core a particular core is unable to offer further performance as it is being fully utilised at that point. At the third point on the graph, we have 32 threads provided by 8 threads over 4 cores. If the benchmark were perfectly scalable wed expect a 10x speedup at best here. At the first point in the graph were using 2 threads, so we have a maximum of $5/2 = 2.5$x speedup from fully utilising a single core and a 4x speedup from the increased core count giving a maximum of 10x. However due to the single control thread we know the scaling of the SAT solver is limited so the maximum speedup may not be possible. Mamba achieves a speedup of 6x which compares favourably with the maximum possible were there no bottlenecks. It also shows once we hit the bottleneck further increases in thread count do not have an adverse impact on performance.

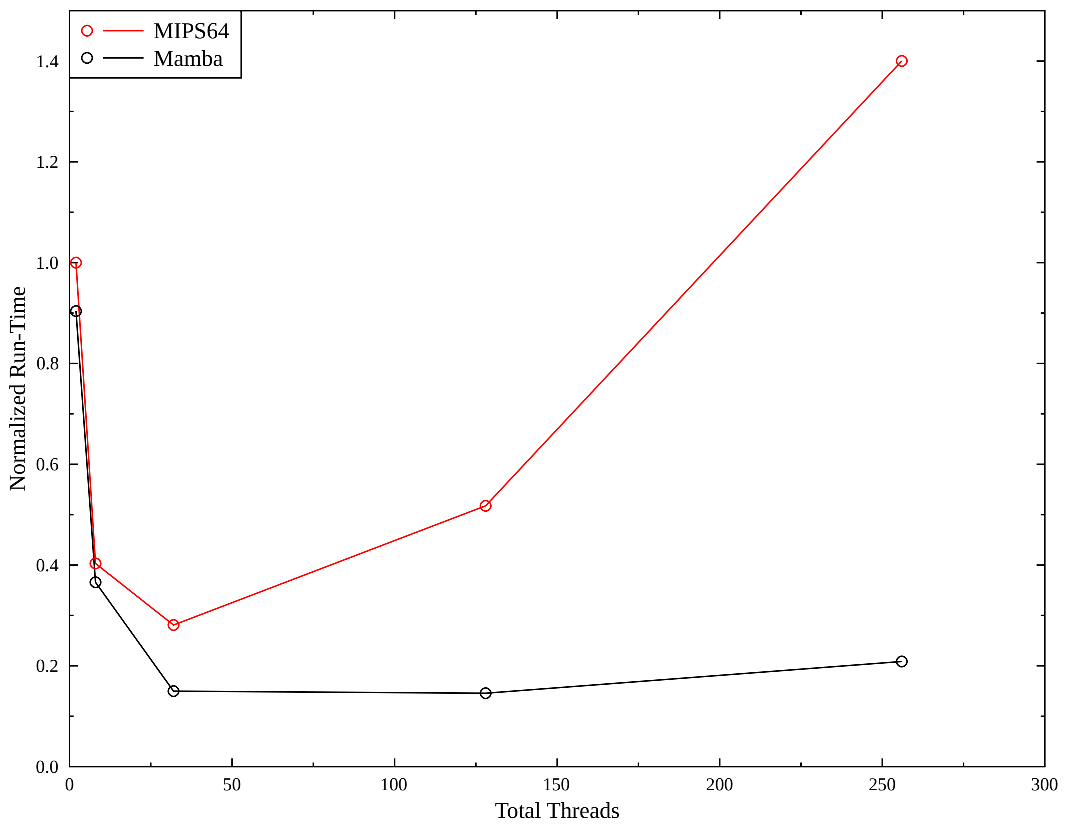


Figure 6.20: Normalised run-time of the SAT solver for a range of total thread counts (threads per core   cores) solving the 'huge' instance. Run-time normalised to the MIPS64 run with 2 threads and 1 core

SAT solver summary

The SAT solver demonstrates that the advantages of the Mamba architecture, illustrated by the micro-benchmarks, can provided real benefit in an actual application. By using presence bit based primitives in the Mamba implementation both performance and scalability are improved compared to the MIPS64 implementation (which is implemented in exactly the same way but without the benefit of presence bits and scheduling guided by them). The SAT solver has an inherent bottleneck in the form of a single control thread, in Mamba when performance has reached the maximum as allowed by this bottleneck further increases in thread and core count did not have the same negative effect that they did in MIPS64. Only when using the more extreme example of the 'medium' instance problem did Mamba exhibit poor scaling when pushing the core and thread count beyond what was needed to gain the best performance.

6.8 Summary

In this chapter we have evaluated the Mamba architecture against a MIPS64 system that lacks Mamba's hardware scheduling and presence bits but it is otherwise identical. Microbenchmarks looking at lock, barrier and FIFO performance show that Mamba is not sensitive to thread count, an equal amount of work split amongst more of less threads results in similar performance, the same was not so of the MIPS64 system. More sophisticated software mechanisms may help close the gap between Mamba and MIPS64, but Mamba can achieve good performance without them. A SAT solver benchmark was presented to demonstrate that the advantages of Mamba shown by the microbenchmarks translated to real gains in an actual application.

Chapter 7

Conclusion

This dissertation presented Mamba, an architecture designed for the challenges of the multi-core era. Based on the idea that to produce a highly scalable system fine-grained techniques with low overheads must be used, Mamba provides lightweight thread scheduling and synchronisation via a hardware scheduler combined with a tagged memory system that is utilised by the scheduler to know when a thread should be scheduled as well as by the threads as a synchronisation mechanism.

Mamba has been fully implemented on FPGA and evaluated against a MIPS64 comparison system, identical to Mamba but lacking the hardware scheduling and presence bit mechanisms. In microbenchmarks of lock and barrier performance and FIFO throughput and latency Mamba has been shown to scale well with increasing thread and core count and has been shown to be insensitive to thread count. Mamba's simple scheduler informed by presence bits is highly effective compared to a similar simple software scheduler that lacks the knowledge that presence bits give to the scheduler.

For a real application, the SAT solver, Mamba continues to show good performance and good scaling compared to the MIPS64 system. Again Mamba was not sensitive to thread count. When the optimum performance as allowed by the application was reached increasing thread count didn't have an immediate detrimental effect, only in the extreme case of the 'medium' instance did performance become poor at higher thread counts. This property of Mamba frees the programmer from having to ensure their application was using the optimum number of threads, unlike in the MIPS64 system where performance quickly degraded from optimum with increasing thread count.

The SAT solver also demonstrated that lightweight low-overhead primitives provided by Mamba do not inherently make software perform and scale better, the bottleneck of the single control thread in the SAT solver eventually limited scaling and performance. What Mamba does do is allow the programmer to use many threads in a fine-grained manner without having to worry about getting the precise number correct to achieve the optimum performance so instead the number of threads must suited to the application (as opposed to the architecture) can be used.

Mamba enables the creation of scalable programs and it has done this without the need for many new architectural features, with a simple scheduler and no extra software support to guide the scheduler scalable programs can be created. Despite being hardware managed thread count is not constrained by some limited hardware resource so a programmer doesn't need to target a particular instance of the architecture to get the best performance from their program. The presence bit mechanism is straight forward and flexible, large number of new instruction and memory semantics are not needed to gain its benefits.

7.1 Comparison with related architectures

In Chapter 3 on fine-grained architecture a few existing systems that use presence bits were reviewed. The Cray XMT [75] supported presence bit based synchronisation for all words in memory but this was implemented differently to Mamba. A thread waiting for a word to become present must spin continuously checking the word before trapping to a software handler if this takes too long. Part of Mamba's strength lies in its ability to immediately stall a thread without the need for spinning when a thread is waiting on a word. The Godson-T [26] implements a similar mechanism but only words that are currently in the cache are tagged with a presence bit, limiting their usefulness.

The LCMT [91] architecture is the most similar to Mamba. The major difference is how it handles multiple threads attempting to load a non-present word. In LCMT this involves an immediate trap to a software handler, whilst Mamba allows the normal flow of the program to continue and to fix the situation itself (e.g., via a notify chain) if it chooses to do so. The LCMT also has more load and store types than Mamba (e.g., store only if non-present). This does allow more flexibility but also adds more complexity to the memory semantics requiring the use of more state for non-present words. In Mamba a non-present word either has a forwarding address or it doesn't in LCMT several other states are introduced (e.g., one which would only send stored data to a forwarding address and leave the word non-present).

7.2 Future work

As clearly demonstrated in the lock and barrier benchmarks when the MIPS64 system began to use its software scheduler, without the benefit of presence bits to guide it, it quickly resulted in bad performance. Mamba's simple hardware scheduler managed to achieve good scaling. It is clear Mamba's methods are useful but it is not clear that the scheduling and context switching system must be entirely hardware based. An interesting avenue for future work would be investigating what parts of the Mamba scheduling system need to be in hardware to get the benefits seen in the evaluation and what can be left in software. One possible design would be a hardware managed ready queue that informs a software scheduler. Replicating the generality of the presence bit mechanism entirely in software is likely to carry a high overhead, however the scheduler itself could have an efficient software implementation, even with the high context switching frequency Mamba currently utilises. A software implementation would also provided far greater flexibility to the programmer, priority levels for threads for example would be a useful feature which would be easily handled if a software scheduler were used.

Every word in Mamba is tagged with a presence bit, however not every word in Mamba uses that presence bit. Some data structures will use them internally for synchronisation but it is likely in any large application that there will be large amounts of memory that do not require the functionality of presence bits. To avoid storing these a virtual memory system could be used, a page would be marked to indicate it uses presence bits, the page tables would also indicate where pages that used presence bits would store the presence bits. Combing this with a hardware managed ready queue would be one possibility for adding Mamba like features to an existing architecture without huge disruption.

The idea of utilising paging so presence bits only take up memory when they're actually required could be extended to the hardware managed ready queue. Initially the queue would be entirely on-chip, which is overflowed the buffer an exception would be triggered and a few pages could be allocated to the ready queue, so the memory utilised by it would be proportional to the maximum number of ready threads present.

Currently when Mamba performs a context switch all register contents are copied, however a thread may not have used all of its registers, some may have contents identical to what is stored in memory, some may never have been read. A lazy context switching system could be created where registers are only copied in the register file from memory when they are used as a source operand. When a context is switched out, only those registers that have been written to would be copied back. This would allow very lightweight threads to be created that run small sections of code with low overhead.

The LCMT architecture briefly discussed in the Fine-grained computation chapter introduces multiple load and store types, allowing actions such as only storing if a word is empty. Whilst such extra functionality increases flexibility it also increases complexity and the extra memory semantics may lead to more compositions of operations having unwanted side effects. Future work should look at the current memory semantics offered by Mamba and see if extra load and store types may be useful and worth the added extra complexity.

The MIPS64 system used as a comparison basis for Mamba is unsophisticated compared to modern CMPs with the benefit of modern operating systems. The simplicity of MIPS64 was necessitated by the need for a comparison system that could be as identical to Mamba as possible and time available meant that this could not include more sophisticated software mechanisms. Future work should look at comparing Mamba to software running on a modern operating system as well as how Mamba could be used by a modern operating system.

7.3 Summary

Overall Mamba has been shown to have great promise, it deals elegantly with many threads and fine-grained synchronisation. The architecture does not constrain the programmer with having to worry about optimising for limits specific to a particular instance of the architecture enabling the creation of scalable fine-grained programs. Mamba is by no means a silver bullet, an application must still be properly constructed to maximise its parallelism but Mamba encourages the creation of such programs.

Appendix A

Mamba: A Scalable Communication Centric Multi-Threaded Processor Architecture

This appendix contains a preprint of the paper that will be presented at the 30th International Conference on Computer Design. It presents Mamba along with a preliminary evaluation. It provides a condensed description of the architecture without the full detail offered in the Mamba architecture chapter.

Abstract

In this paper we describe Mamba, an architecture designed for multi-core systems. Mamba has two major aims: *(i)* make on-chip communication explicit to the programmer so they can optimize for it and *(ii)* support many threads and supply very lightweight communication and synchronization primitives for them. These aims are based on the observations that: *(i)* as feature sizes shrink, on-chip communication becomes relatively more expensive than computation and *(ii)* as we go increasingly multi-core we need highly scalable approaches to inter-thread communication and synchronization. We employ a network of processors where a given memory access will always go to the same cache, removing the need for a coherence protocol and allowing the program explicit control over all communication. A presence bit associated with each word provides a very lightweight, fine-grained synchronization primitive. We demonstrate an FPGA implementation with micro-benchmarks of standard spinlock and FIFO implementations and show that presence bit based implementations provide more efficient locking, and lower latency FIFO communications compared to a conventional shared memory implementation whilst also requiring fewer memory accesses. We also show that Mamba performance is insensitive to total thread count, allowing the use of as many threads as desired.

A.1 Introduction

In the last few years we have seen the rise of the Chip Multiprocessor (CMP). With clock speeds staying static but feature sizes still shrinking exploiting thread level parallelism (TLP) along with instruction level parallelism (ILP) is the natural way to gain further performance. Given a constant die size with shrinking feature size, the chip area reachable in a single cycle also shrinks. This complicates the design of a single large core that uses the increasing availablity of gates to extract more ILP. Using that area to instead have a multitude of smaller, simpler cores, increasing core number, rather than core complexity, is preferable and the costlier global cross-chip communcations can be made architecturally explicit, allowing software to optimize for it [86].

Most CMPs utilize shared memory along with a cache coherency protocol to allow communication between separate threads and cores. The cache coherency protocol is responsible for deciding when to communicate between cores and the programmer can only indirectly influence this communication by choosing what data is shared between threads. Much effort has been made to optimize coherency protocols to reduce needless communication ([28], [7]). Furthermore, in order to achieve good performance sequential consistency is not maintained, and a more relaxed consistency model used instead [1]. This breaks that idea that a shared memory system with cache coherency presents a familiar memory model to the programmer.

With increasing core counts [102] software needs to be capable of achieving performance that scales with the number of cores. Amdahl's law shows that this is limited by the time spent synchronizing and in critical sections [31]. So for a program to scale well with core count it needs to have as much TLP as possible, and for each thread to spend as little time as possible synchronizing and in critical sections. This suggests a programming model that provides very light-weight threads and synchronization primitives to go with them will enable software that scales well with an increasing number of cores.

This paper presents Mamba, an architecture that

i) Gives explicit control over communication, so software can be optimized to reduce it

ii) Provides a light-weight threading and synchronization model.

We introduce Mamba in section 2, discuss some software techniques for it in section 3, review related work in section 4, present a evaluation of an MCS Lock and FIFO queue implementation in section 5, conclude in section 6 and consider future work in section 7.

A.2 Mamba Architecture

A Mamba system consists of a network of nodes. Each node contains a simple in-order RISC core, a cache and a network-on-chip (NoC) router. Each node is allocated an area of global physical address space which is termed that node's local address space; any other addresses are in a remote address space. When a node's core accesses local address space it goes to the local cache. Remote address space is accessed from remote cache via the network. Provided the network retains ordering of messages between nodes we can ensure sequential consistency within a particular local address space.

It should be emphasised that the programmer's view of memory is fully coherrent. Whilst there is no explicit cache coherency system, enforcing that a given area of memory must always be cached in the same cache ensures there is no duplication which is effectively acting as a very simple cache coherency protcol.

Every 64-bit memory word in a Mamba system has an associated presence-bit (also known as a full/empty bit). This is used as a basic primitive for thread to thread communication and synchronization. A thread attempting to read a non-present location will be stalled and descheduled until that location becomes present. Each register also has an associated presence bit, when a thread attempts to use a non-present register it stalls until the register becomes present. Presence bits are stored in main memory at the top of each node's local address space, this leaves a gap in the address space that cannot be used for data storage, however this could be hidden by a virtual memory system.

A node supports hardware threading and scheduling. The register file (RF) in a node's core can hold the registers of eight separate threads. Every cycle it will issue an instruction from a different thread, scheduling the threads within the register file in a round robin manner. A thread is represented in memory by an activation frame (AF), this is simply the contents of the thread's registers, its program counter (PC) and a status word. It fits in a 256-byte (32 x 64-bit words) block.

There is a special store instruction (store doubleword to AF, **SDA**) that causes a node to check the presence bits of the AF being stored to and if they are all set it places the AF on a queue of ready nodes. The node has a simple round-robin scheduler that switches contexts from the RF back into memory and switches an AF from the ready queue from memory into the RF. A context switch occurs when an active thread's quantum expires.

Each node has three separate caches. An instruction cache, a data cache and a presence bit cache. The instruction cache does not obey the restriction that a particular address can only live in a particular cache, so code may be replicated across instruction caches. A seperate presence bit cache is used as it allows checking of an entire AFs worth of presence bits without looking at multiple cache lines (which would be the case if presence bits were stored along with words in the data cache).

When a core executes a load or a store instruction it generates a memory request, this will be sent directly to the local cache or out to the network depending upon the address. Upon executing a load a core will clear the presence bit of the register which is the load's destination, so if a thread attempts to use that register before the load has completed it will stall.

There are two major request types, load and store:

Load (Figure A.1) has an address (A) to load from and a return address (R). The return address is where the response from the load should be sent. As each thread is represented by an AF, each register has a memory address so the return address is the address of the register that was the load's destination. When a load request is received the presence bit of the corresponding word is checked, if it is:

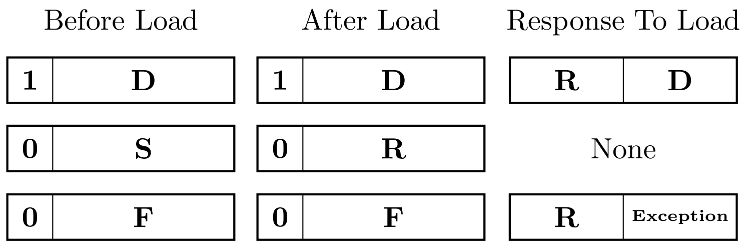


Figure A.1: The possible actions on a load request at a particular word. D is the data stored in the word, S is the sentinel value, R is the return address of the load and F is an already existing return address. **0** refers to a non-present word, **1** to a present word.

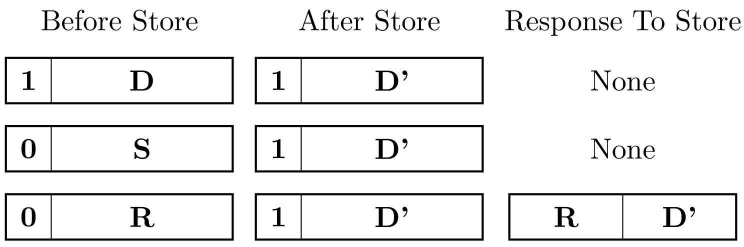


Figure A.2: The possible actions on a store request at a particular word. D is the existing data stored at the word, D' is the new data the store request is writing, S is the sentinel value and R is a return address. **0** refers to a non-present word, **1** to a present word.

Present — An immediate data response is sent to the return address with the contents of the word.

Non Present — The contents of the word are checked, if it is a sentinel value, the return address of the load is written into the word and nothing else is done. If the sentinel value is not there (The sentinel value is a particular invalid return address), then some other load request has already written its return address into this word and an exception response is immediately sent.

Store (Figure A.2) has an address (A) to store to and the data to store (D'). When a store request is received the presence bit of the corresponding word is checked, if it is:

Present — The contents of the word are overwritten with the new data, nothing else is done.

Non Present — The presence bit is set and the current word contents are checked. If the sentinel value is there, nothing further is done. Otherwise a load return address is there and a load response with the store data is generated to that address.

Upon receiving a data response, the AF portion of the address is checked. If the AF is in the RF then the corresponding register is updated with the data from the response and the register's presence bit is set. If the AF has been swapped out to memory then the corresponding word within the AF is updated and its presence bit set. The presence bits of all words within the AF are checked and if they are all set the AF is placed at the back of the ready queue.

The mechanism above allows many communication styles. For simple producer single consumer communication the presence bit of a particular word can be cleared, at some point a thread (the consumer) loads from that word. When the thread tries to use the result of that load it will stall (as the word was non-present so no response is received and thus the register holding the result is still marked as non-present). At a later point another thread (the producer) will write to the non present word causing a data response to be sent to the consumer's AF. Either the consumer will still be in the register file so upon receiving the response can begin execution again immediately or the scheduler will have swapped it out in which case the data response will cause all words within the AF to be present so the consumer will be placed at the back of the ready queue.

If we add a second consumer which attempts to read a non-present word an immediate exception response is sent. So when the producer eventually writes to the non-present word only the first consumer will receive the data (see Figure A.3). There are two possible solutions. If the number of consumers is fixed and known ahead of time the single producer multiple consumer situation can be turned into multiple single producer single consumer situations. If this is not possible, or undesirable, a software solution described in the software techniques section can be employed.

To enable this software mechanism a second load instruction is added. When the consumer receives an exception response from a load there are two possible things that can happen:

An exception is triggered in the consumer's thread and it jumps to a handler

A bit is set in an exception status register corresponding to the destination register the load was intended for and execution continues

The two separate load instructions allow the programmer to choose how the exception response is dealt with. The first (**LD** the standard load instruction) will cause the first action to occur on an exception response and the other (**LDNR**) will cause the second action to occur. So **LD** is used when we either expect the word will always be present or that if non-present that only a single thread will attempt to read it. **LDNR** will be used when we expect multiple consumers of a non-present word. After executing the load the software must check the exception status before using the result of the load. If an exception was received then the software must take appropriate action (precisely what this is depends upon the software, a possible scenario is described in the software techniques section).

If the consumer's AF is swapped out at the time of receiving the read exception the status word in the AF is updated to indicate the exception and the appropriate action is taken when the AF gets swapped back.

A.2.1 FPGA Implementation

To experiment with the Mamba architecture we have produced a fully functional FPGA implementation. It executes a modified subset of the MIPS64 ISA. MIPS64 was chosen

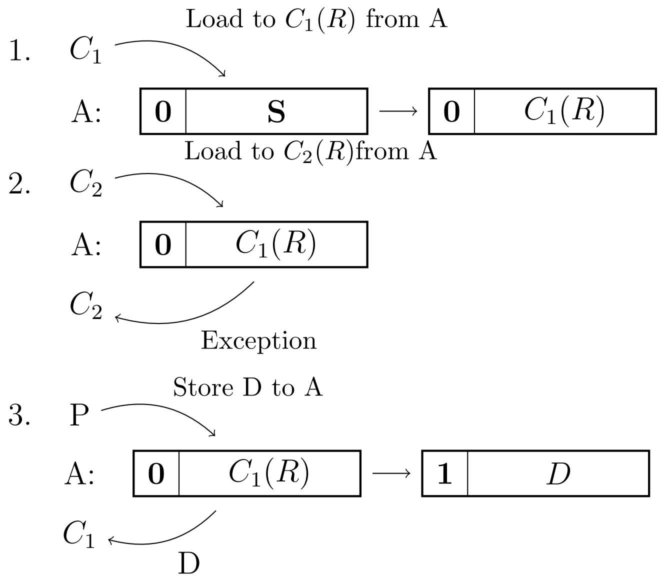


Figure A.3: Producer Consumer. $C\ _n(R)$ represents the address of register R in the AF C_n. In a single consumer situation only steps 1. and 3. will occur. In a multiple consumer situation all 3 steps will occur.

due to its ease of implementation and existing compiler tool chain. For a comparison system we have implemented a plain MIPS64 core which is implemented very similarly to Mamba but lacks presence bits and hardware thread scheduling. It has 8 fixed hardware threads that like Mamba are implemented with 8 separate register files and an instruction for a different thread being issued every cycle, but unlike Mamba there is no hardware thread switching or hardware scheduling system. To allow implementation of concurrency primitives a CAS (compare-and-swap) operation was added. A software thread scheduler was also written, this functions exactly as the Mamba thread scheduler does (pre-emptive, round-robin, fixed quantum) but is implemented entirely in software.

We use Altera Stratix IV FPGAs on a DE4 board. Multiple DE4 boards can be connected with high speed serial links to increase the number of cores available. The current setup uses a single board which can hold four Mamba cores on the FPGA. We are working on building a larger system with 64 DE4 boards connected in a grid topology.

A.3 Software Techniques

Another use of the presence bits is as a binary sempahore to guard access to the associated word. This is implemented using a third load instruction (see table A.1), the vacating load, with mnemonic **VL** (chosen as it's an existing instruction within the MIPS64 instruction set, however the semantics are not the same). The vacating load checks the presence bit of the word it is loading. If it is set it clears the bit and sends back a data response. If the bit is already clear it sends back an exception response (Setting a bit in the read exception status register).

This can be used to atomically update a word. Say the word contains a counter, a thread

Table A.1: The three kinds of load, their mnemonics and what they do to the word they're loading

Mnemonic	Action if present	Action if not present
LD	Sends data response to return address	Write return address into word if sentinel present, otherwise sends exception response, triggering exception handler in thread
LDNR	Sends data response to return address	Write return address into word if sentinel present, otherwise sends exception response, setting read exception flag for destination register
LL	Clear presence and sends data response to return address	Sends exception response, setting read exception flag for destination register

can use a **VL** to get the counter, increment it, and then store it back. Between the load and the store the presence bit is unset so any other thread which attempts to load it will either stall waiting for the data to be written or will receive an exception response back. Any thread seeking to also atomically update the counter will spin doing repeated **VL**s until it successfully gets the word.

Ideally if a thread is unable to continue because a word it requires is not present it should stall and be descheduled rather than spin. This is dealt with by the architecture as described in the mamba architecture section for the producer single consumer situation but otherwise this can be accomplished via a lightweight construct we call a 'notify chain'. These are similar in structure to MCS queue based spin-locks [73] with one crucial difference. Rather than spinning on a particular value waiting for it to become 1, the presence bit of that value is cleared, so when the thread loads it and attempts to use it, it stalls and is descheduled. It is then woken by another thread writing into that value.

In more detail, the notify chain is a singly-linked list of two word nodes. The first word (the wait word) initially has its presence bit cleared, the second word is a pointer to the next node in the chain. When a thread wishes to wait on a notify chain it first constructs a node within its local memory space and then atomically inserts itself into the tail of the chain. Finally it loads the wait word (which has a cleared presence bit) and immediately uses the result of the load, which has the effect of causing the thread to stall until a store sets the presence bit of the wait word. When a thread wishes to notify the notify chain it simply writes to the wait word of the head of the chain. When a thread is woken up in this manner it could either immediately wake the next thread in the chain or wake the next thread later (depending on if a notify next, or notify all behaviour is desired). As each thread creates its notify node within its local space the store request which triggers the thread wakeup will go directly from the thread that triggers the wakeup to the thread that receives the wakeup, which is the minimum communication required for such an action.

Atomic insertion into the chain tail is accomplished using **VL**. To insert a notify node into the tail a thread simply **VL**s the tail pointer. It updates the tail's node's (if there is one) next to point to its own node then writes a pointer to its own notify node as the new tail pointer. When a thread wants to notify the chain it must also **VL** the tail pointer. This is for two reasons:

1. It avoids a lost wakeup problem where a thread notifies the thread owning the tail node before the tail node's next is updated to the new tail during an insert
2. It needs to check if it is the tail node and update the tail pointer to NULL if it is

This general notify chain mechanism can be used for a variety of tasks such as:

1. As part of a queue based lock, where a thread will stall if it's unable to take the lock and woken up by the thread in front of it when its done with the lock.
2. As part of a barrier, threads can wait on a notify chain until something notifies them all that they can proceed (a counter could keep a count of threads in the chain, when a thread adds itself to the chain it will check the counter, if its hit a certain number it will notify all, otherwise it will increment the counter and wait).
3. As part of a producer multiple consumer situation. A consumer could try to load a word with **LDNR** if it receives an exception response it will wait on a notify chain. The producer will notify this chain when it stores to the word, the new word value can be used as the value written to the wait word so the waiting threads receive it immediately on wakeup.

Whilst the consistency model employed by Mamba requires a network that preserves the ordering of messages between two nodes, is it not essential that this is always the case. The current software targetted at the architecture relies on this fact but it could be rewritten to take the reordering of messages into account.

A.4 Evaluation

To test the FPGA implementation of the system we have written a series of micro-benchmarks, two are presented here. Each micro-benchmark has been implemented on the Mamba and MIPS64 systems so results can be compared.

As the Mamba and MIPS64 systems both run on the same FPGA, using the same caches, memory controllers and interconnect architecture, details such as memory latency and the raw throughput and latency of the network are the same in both cases so are not considered here.

A.4.1 MCS Lock Benchmark

The MCS Lock [73] is a standard way of implementing a scalable spin-lock. Any thread wishing to enter the lock first constructs a node, it then attempts to atomically add that

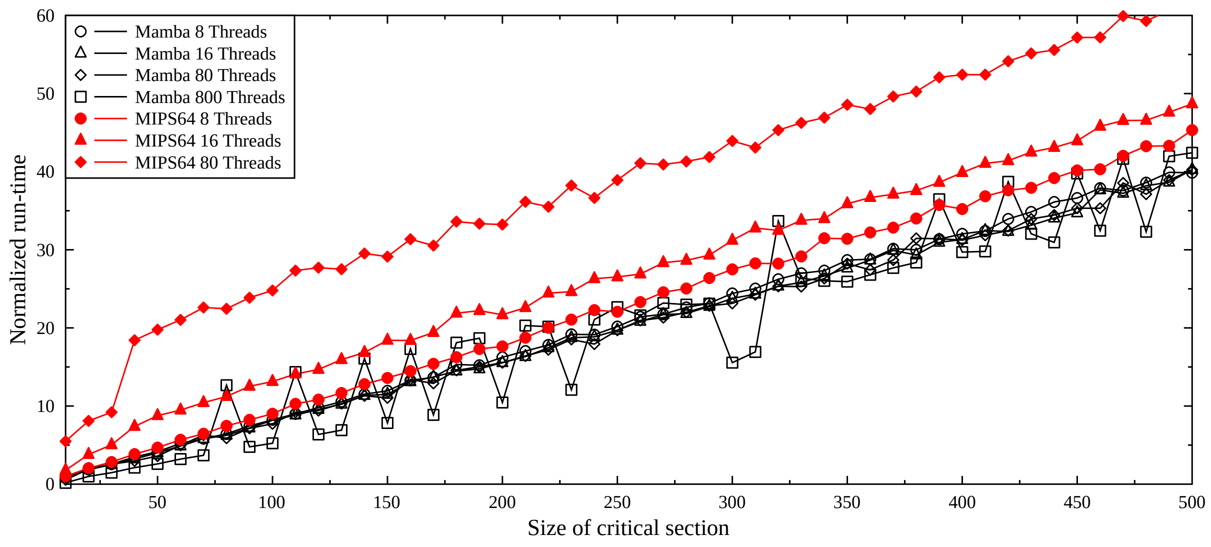


Figure A.4: Run-time for the MCS Lock Benchmark with increasing work size in the critical section, run-time normalized to MIPS64 run-time with smallest critical section size. Benchmark run over 4 cores in all cases

node to a queue. If the queue is empty the thread gains the lock, enters the critical section and places its node as the only element in the queue. If the queue is not empty it places its node at the tail of the queue and spins on its node waiting for a lock field to turn from false to true. When a thread releases the lock it checks its node to see what is next in the queue, if there is a node there it sets its lock field to true giving the next thread the lock and allowing it into the critical section. As the node is constructed in local storage the spin does not generate any needless remote memory accesses.

We have implemented the MCS lock on MIPS64 using the CAS primitive as described in [73]. In Mamba a notify chain as described above is used. When a thread attempts to acquire a lock it simply waits on the notify chain, when it does this the thread will be descheduled until woken. During this period it won't perform any operation, and in particular any memory operations, unlike the MIPS64 version which must spin continuously reading the lock field of its node.

All four cores available in the FPGA implementation are used, each contains a single shared area protected by an MCS lock. Every thread randomly chooses a shared area to access and attempts to acquire the MCS lock. In the shared area is an array of integers, once a thread has acquired this MCS lock it sums these integers and writes the result into the shared area, it then replaces the integers with new randomly generated values. The point of this is so the thread performs some operation on the data protected by the lock as well as doing its own local computations whilst in the critical section to simulate the kind of work that would be done in a real application. Once a thread has left the lock it loops round and repeats the process again with another randomly selected shared area. This repeats for a number of rounds.

Once all threads have done all their rounds the number of integers summed and generated within the shared area is increased and the benchmark is run again. The number of cycles taken for the benchmark to run for different critical sections size (as measured by the number of integers summed and generated in the critical section) is measured for the MIPS64 implementations and Mamba implementations with 8, 16, 80 and 800 threads per core. With each increase in thread count the number of rounds a thread does is correspondingly scaled down so the total amount of work done is the same (100 rounds per thread with 8, 50 rounds per thread with 16, etc). The cycle count of each run is normalized to that of the 8 thread MIPS64 run with the smallest amount of work (10 numbers summed and generated).

The run-time of the MCS lock benchmark with increasing work in the critical section can be seen in Figure A.4. They show that with 8 threads MIPS64 is only slightly worse than Mamba (betwen 5% - 16% worse), however with Mamba thread count does not have any particular impact on the benchmark performance. Whilst at higher thread counts the variation of results from the trend is greater, the benchmark still scales equally well amongst all thread counts. For MIPS64 it gets steadily worse as thread count is increased. The results for 800 threads would not fit on the graph, with the smallest critical section size run-time was 48 worse compared to the 8 thread version and 16 worse with a critical section size of 100.

Also of note are the average memory operations per core. A memory operation is defined as any read or write that occurs in a node's data cache. A Mamba thread can sleep and be woken by its node in the MCS lock queue being written to so it does not generate any memory operations spinning whilst waiting to be notified. MIPS64 on the other hand has all threads continuously generating memory operations even if they're not doing any useful work. As a result MIPS64 produces between 14 and 16 as many memory operations as Mamba during the benchmark run in the 8 thread run. Interestingly MIPS64 produces less memory operations in the 16 thread run compared to its own 8 thread run, but this is still between 7 and 10 more memory operations compared to the Mamba 16 thread run. It produces more memory operations in its 80 thread run compared to its 16 thread run and between 10 and 22 more than the Mamba 80 thread run.

A.4.2 FIFO Queue Benchmark

A simple FIFO queue implementation was written based upon [58]. The queue is based around a ring buffer, with read and write pointers pointing to the item at the top of the queue, and the next free slot in the buffer respectively. Provided we guarantee that at most one thread enqueues and one thread dequeues at any given time nothing beyond normal load and store operations are needed (i.e. we do not need CAS). For MIPS64 this has been implemented exactly as [58], for Mamba a simple modification was made. The FIFO ring buffer starts with all its presence bits unset. When dequeuing, a thread checks the size of the queue by looking at the read and write pointers; if it is empty it spins waiting until an item appears. In Mamba we simply directly read the slot the read pointer points to. If the queue is empty then this points to the slot the next item will be written to and so will be non-present and the thread will sleep until an item is added. When we have dequeued an item we clear the presence bit on the ring buffer slot it came from and then increment the read pointer (ensuring the enqueueing thread will not write to the slot until we have cleared the presence bit).

We have measured the throughput and the latency of the Mamba and MIPS64 implementations of this queue. One core executes all the consumer threads, these each had their own FIFO queue in their local address space. Paired with a consumer thread was a producer thread, which was located on another core. The producer thread simply sent a sequence of increasing integers.

The number of cores used to produce was varied between 1-3 with 1 or 2 threads on each core. The 3 producer core arrangement was also measured with 4, 8, 16 and 32 threads per core to test scaling ability. The first core executes all the consumer threads so ran between 1 and 96 threads depending on the number of producer threads and cores. The measurements for each thread were averaged and normalized to the result for the MIPS64 implementation with 1 thread and 1 core.

Latency results can be seen in Figure A.5, they show that the Mamba implementation gives between 75% and 641% better latency, achieving the best latency improvement in the 32 threads per producer core with 3 producer cores arrangement. They also show that the Mamba implementation scales far better than MIPS64 with increasing thread count. The bars for 16 and 32 producer threads per core would not fit on the graph for MIPS64 and are 17.7 and 63.1 respectively compared to 4.3 and 9.8 for Mamba. Below 4 threads per core the performance gap remains roughly similar with Mamba between 75% and 85% better.

As the implementation of MIPS64 and Mamba both utilise the exact same interconnect architecture the latency introduced by the network isn't taken into account as it will be the same in both cases, so it is not essential that is is separated from the latency introduced by the differing FIFO implementations

Throughput results can be seen in Figure A.6, they show that the Mamba implementation gives between 13% and 246% better throughput. Again the Mamba implementation scales far better than MIPS64 with increasing thread count. Below 8 threads per core the performance gap remains roughly similar with Mamba between 13% and 27% better.

The Mamba implementation shows good scaling with throughput, if the measured throughput is multiplied by the number of threads producing per core we find it reaches a peak of 3.19 at 8 threads producing per core with 3 producer cores and only reduces to 2.61 for 32 threads producing per core with 3 producer cores.

A.5 Related Work

Godson-T [26] is also a MIPS based multicore architecture that utilizes presence bits. However unlike Mamba they are only present in cache-lines and are only used when those cache-lines are configured as a scratch-pad memory. They may be used to allow fine-grained synchronization between currently running threads but they don't interact directly with thread scheduling so cannot be used to implement a notification system that wakes a thread when data is ready. Separate hardware mechanisms are used to implemented fine-grained locking and barriers. The Cray XMT [75] building on earlier work done by the J-Machine [80] and HEP [93], utilizes hardware multithreading and has a presence bit per word in memory, though when a presence bit is not of the desired state

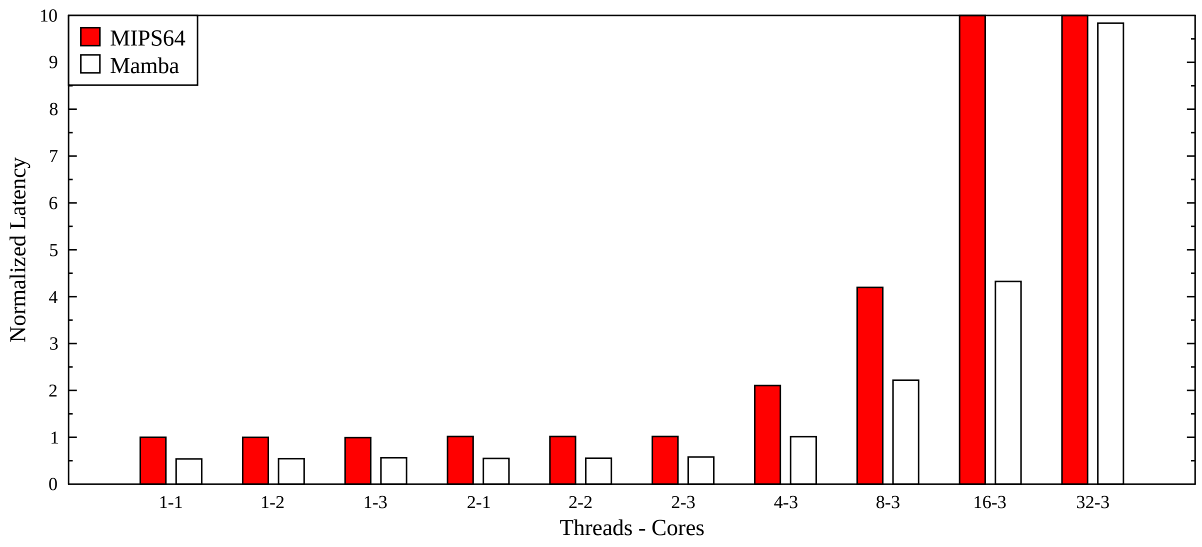


Figure A.5: Normalized latency for the FIFO Queue Benchmark, for differing thread and core numbers, where t-c on the axis labels refers to t threads per producer core and c cores producing

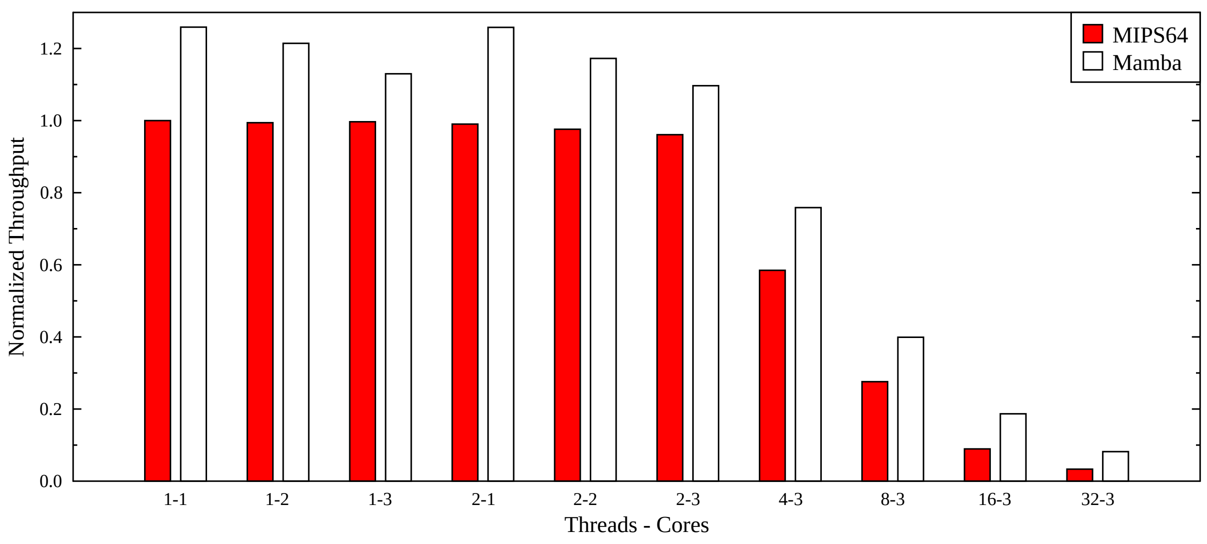


Figure A.6: Normalized throughput for the FIFO Queue Benchmark, for differing thread and core numbers, where t-c on the axis labels refers to t threads per producer core and c cores producing

a thread simply spins until it is, eventually trapping to a software handler if this takes too long. Mamba builds on earlier concepts from Cambridge [77].

The Intel x86 ISA [48] offers an MWAIT instruction, which combined with a MONITOR instruction allows a program to wait for a write to a particular address range. MWAIT can be used to optimise anything relying on repeated polling of a memory location, much like presence bits in Mamba can be used. However the MWAIT mechanism is not equivalent to the presence bit mechanism. It only provides a hint to the processor that it can enter an implementation defined optimized state. After executing an MWAIT memory must be checked again and it has no direct interaction with a scheduler. In contrast with Mamba, once a load of an initially empty word completes you can be sure that a value has been written to the word. When a word needed by a thread becomes available the thread immediately gets added to the scheduler's ready queue. As an MWAIT would only be used when a processor is idle to construct a similar mechanism without presence bits would require a thread to notify the scheduler about certain memory operations, this would carry far higher overhead effecting performance and scalability.

A.6 Conclusion

We have demonstrated and prototyped Mamba on FPGA, a processor architecture that makes communication explicit, provides scalable hardware threading and a fine-grained synchronisation and notification mechanism based on presence bits. With minor modifications to an MCS lock implementation, replacing spinning on a local word to waiting for a local word to become present we achieved better performance as well as dramatically reducing the number of memory accesses required. With minor modifications to a FIFO queue implementation latency and throughput was greatly improved.

Mamba's threading model provides performance that is insensitive to thread count. For the MCS lock benchmark thread count was of little consequence to run-time, for FIFO communication total throughput did not decline much with increasing thread count. This frees a programmer from having to carefully consider the number of threads to use for a particular task, or for the need to change this for different numbers of cores.

We conclude that Mamba is a promising architecture for the future of increasingly multi-core systems. Existing concurrency primitives can be simply adapted to utilize presence bits achieving performance gains and threads can be employed as the programmer desires.

A.7 Future Work

One interpretation of the results would be that the hardware scheduler of Mamba brings about the majority of the benefits seen and the presence bit mechanism is not required. However it is the presence bit mechanism that allows a particular thread to wait for a word to become present without needless polling and it allows the scheduler to know what threads are ready. As stated above a purely software version of a similar mechanism would add overhead preventing it from being used at such a finely grained level. Though a hardware managed ready queue with a software scheduler or hybrid software/hardware scheduler is a design point worth exploring in future work.

The presented architecture lacks a virtual memory system. A key design point of a virtual memory system for Mamba is whether a word's physical or virtual address specifies which node it belongs to. If a virtual address is used there are two issues. The first is aliasing, two separate virtual address may map to the same physical address breaking coherency and the virtual address mapping may distort or totally obscure the relation between addresses and word location, hiding the communication costs the architecture attempts to make explicit. Using a physical address would not present these issues but using a virtual address adds functionality, for example it could be used to create a data and thread migration system where the node holding a particular piece of data or a thread could be altered without disturbing the memory address needed to access it. The aliasing issue is an existing problem already dealt with by operating systems and NUMA systems need to allocate pages with due regard to the locality of memory so solving these issues in the specific case of Mamba should be not problematic.

Acknowledgments

The authors would like to acknowledge Arnab Banerjee for implementing the interconnect architecture used by the Mamba system and would like to thank Paul Fox, Timothy Jones and Theo Markettos for their valuable feedback.

Bibliography

[1] S.V. Adve and K. Gharachorloo. Shared memory consistency models: a tutorial. *Computer*, 29(12):66 –76, December 1996.

[2] A. Agarwal, R. Simoni, J. Hennessy, and M. Horowitz. An evaluation of directory schemes for cache coherence. In *Proceedings of the 15th Annual International Symposium on Computer architecture*, ISCA '88, pages 280–298, Los Alamitos, CA, USA, 1988. IEEE Computer Society Press.

[3] Vikas Agarwal, M. S. Hrishikesh, Stephen W. Keckler, and Doug Burger. Clock rate versus IPC: the end of the road for conventional microarchitectures. In *Proceedings of the 27th annual international symposium on Computer architecture*, ISCA '00, pages 248–259, New York, NY, USA, 2000. ACM.

[4] Altera Corporation. *Stratix IV Device Handbook.* Number SIV5V1-4.6. September 2012.

[5] Gene M. Amdahl. Validity of the single processor approach to achieving large scale computing capabilities. In *Proceedings of the April 18-20, 1967, spring joint computer conference*, AFIPS '67 (Spring), pages 483–485, New York, NY, USA, 1967. ACM.

[6] C.S. Ananian, K. Asanovic, B.C. Kuszmaul, C.E. Leiserson, and S. Lie. Unbounded transactional memory. In *High-Performance Computer Architecture, 2005. HPCA-11. 11th International Symposium on*, pages 316 – 327, Feburary 2005.

[7] Anoop Gupta, Wolf-dietrich Weber, and Todd Mowry. Reducing Memory and Traffic Requirements for Scalable Directory-Based Cache Coherence Schemes. In *In International Conference on Parallel Processing*, pages 312–321, 1990.

[8] Joe Armstrong. *Programming Erlang: Software for a Concurrent World.* Pragmatic Bookshelf, 2007.

[9] Assaf Shacham, Keren Bergman, Senior Member, and Luca P. Carloni. Photonic Networks-On-Chip for Future Generations of Chip Multiprocessors. *IEEE Trans. Computing*, page 1260, 2008.

[10] Arnab Banerjee. Communication flows in power-efficient Networks-on-Chips. Technical Report UCAM-CL-TR-786, University of Cambridge, Computer Laboratory, August 2010.

[11] Nick Barrow Williams, Christian Fensch, and Simon Moore. Proximity coherence for chip multiprocessors. In *Proceedings of the 19th international conference on Parallel architectures and compilation techniques*, PACT '10, pages 123–134, New York, NY, USA, 2010. ACM.

[12] Scott Beamer, Chen Sun, Yong-Jin Kwon, Ajay Joshi, Christopher Batten, Vladimir Stojanović, and Krste Asanović. Re-architecting DRAM memory systems with monolithically integrated silicon photonics. In *Proceedings of the 37th annual international symposium on Computer architecture*, ISCA '10, pages 129–140, New York, NY, USA, 2010. ACM.

[13] Robert D. Blumofe and Charles E. Leiserson. Scheduling multithreaded computations by work stealing. *J. ACM*, 46(5):720–748, September 1999.

[14] Jason F. Cantin, Mikko H. Lipasti, and James E. Smith. Improving Multiprocessor Performance with Coarse-Grain Coherence Tracking. *SIGARCH Comput. Archit. News*, 33(2):246–257, May 2005.

[15] Calin Cascaval, Colin Blundell, Maged Michael, Harold W. Cain, Peng Wu, Stefanie Chiras, and Siddhartha Chatterjee. Software Transactional Memory: Why Is It Only a Research Toy? *Queue*, 6(5):46–58, September 2008.

[16] Luis Ceze, James Tuck, Pablo Montesinos, and Josep Torrellas. BulkSC: bulk enforcement of sequential consistency. In *Proceedings of the 34th annual international symposium on Computer architecture*, ISCA '07, pages 278–289, New York, NY, USA, 2007. ACM.

[17] S. Chaudhry, R. Cypher, M. Ekman, M. Karlsson, A. Landin, S. Yip, H. Zeffer, and M. Tremblay. Rock: A High-Performance Sparc CMT Processor. *Micro, IEEE*, 29(2):6 –16, March 2009.

[18] Alex C. Chow, Gordon C. Fossum, and Daniel A. Brokenshire. *A Programming Example: Large FFT on the Cell Broadband Engine*. IBM, May 2005.

[19] C. Clauss, S. Lankes, P. Reble, and T. Bemmerl. Evaluation and improvements of programming models for the Intel SCC many-core processor. In *High Performance Computing and Simulation (HPCS), 2011 International Conference on*, pages 525 –532, July 2011.

[20] G. Contreras and M. Martonosi. Characterizing and improving the performance of Intel Threading Building Blocks. In *Workload Characterization, 2008. IISWC 2008. IEEE International Symposium on*, pages 57 –66, September 2008.

[21] William J. Dally and Brian Towles. Route packets, not wires: on-chip inteconnection networks. In *Proceedings of the 38th annual Design Automation Conference*, DAC '01, pages 684–689, New York, NY, USA, 2001. ACM.

[22] W.J. Dally and B. Towles. *Principles and Practices of Interconnection Networks*. The Morgan Kaufmann Series in Computer Architecture and Design. Morgan Kaufmann Publishers, 2003.

[23] Peter Damron, Alexandra Fedorova, Yossi Lev, Victor Luchangco, Mark Moir, and Daniel Nussbaum. Hybrid transactional memory. In *Proceedings of the 12th international conference on Architectural support for programming languages and operating systems*, ASPLOS-XII, pages 336–346, New York, NY, USA, 2006. ACM.

[24] Martin Davis, George Logemann, and Donald Loveland. A machine program for theorem-proving. *Commun. ACM*, 5(7):394–397, July 1962.

[25] Dave Dice, Ori Shalev, and Nir Shavit. Transactional locking II. In *Proceedings of the 20th international conference on Distributed Computing*, DISC'06, pages 194–208, Berlin, Heidelberg, 2006. Springer-Verlag.

[26] Dongrui Fan, Hao Zhang, Da Wang, Xiaochun Ye, Fenglong Song, Guojie Li, and Ninghui Sun. Godson-T: An Efficient Many-Core Processor Exploring Thread-Level Parallelism. *Micro, IEEE*, 32(2):38 –47, March 2012.

[27] A. Duller, G. Panesar, and D. Towner. Parallel Processing-the picoChip way. *Communicating Processing Architectures*, 2003:125–138, 2003.

[28] S. J. Eggers and R. H. Katz. Evaluating the performance of four snooping cache coherency protocols. In *Proceedings of the 16th annual international symposium on computer architecture*, pages 2–15, New York, NY, USA, 1989. ACM.

[29] Alexandre E. Eichenberger, Kathryn O'Brien, Kevin O'Brien, Peng Wu, Tong Chen, Peter H. Oden, Daniel A. Prener, Janice C. Shepherd, Byoungro So, Zehra Sura, Amy Wang, Tao Zhang, Peng Zhao, and Michael Gschwind. Optimizing Compiler for the CELL Processor. In *Proceedings of the 14th International Conference on Parallel Architectures and Compilation Techniques*, PACT '05, pages 161–172, Washington, DC, USA, 2005. IEEE Computer Society.

[30] Natalie D. Enright Jerger, Li-Shiuan Peh, and Mikko H. Lipasti. Virtual tree coherence: Leveraging regions and in-network multicast trees for scalable cache coherence. In *Proceedings of the 41st annual IEEE/ACM International Symposium on Microarchitecture*, MICRO 41, pages 35–46, Washington, DC, USA, 2008. IEEE Computer Society.

[31] Stijn Eyerman and Lieven Eeckhout. Modeling critical sections in Amdahl's law and its implications for multicore design. In *Proceedings of the 37th annual international symposium on Computer architecture*, ISCA '10, pages 362–370, New York, NY, USA, 2010. ACM.

[32] Michael J. Flynn. Some Computer Organizations and Their Effectiveness. *Computers, IEEE Transactions on*, C-21(9):948 –960, September 1972.

[33] Keir Fraser and Tim Harris. Concurrent programming without locks. *ACM Trans. Comput. Syst.*, 25(2), May 2007.

[34] James R. Goodman. Using cache memory to reduce processor-memory traffic. In *Proceedings of the 10th annual international symposium on Computer architecture*, ISCA '83, pages 124–131, New York, NY, USA, 1983. ACM.

[35] Naga K. Govindaraju, Brandon Lloyd, Yuri Dotsenko, Burton Smith, and John Manferdelli. High performance discrete Fourier transforms on graphics processors. In *Proceedings of the 2008 ACM/IEEE conference on Supercomputing*, SC '08, pages 2:1–2:12, Piscataway, NJ, USA, 2008. IEEE Press.

[36] Cary Gunn. CMOS Photonics for High-Speed Interconnects. *IEEE Micro*, 26(2):58–66, 2006.

[37] Jungwoo Ha, Matthew Arnold, Stephen M. Blackburn, and Kathryn S. McKinley. A concurrent dynamic analysis framework for multicore hardware. In *Proceedings of the 24th ACM SIGPLAN conference on Object oriented programming systems languages and applications*, OOPSLA '09, pages 155–174, New York, NY, USA, 2009. ACM.

[38] Tim Harris, Mark Plesko, Avraham Shinnar, and David Tarditi. Optimizing memory transactions. In *Proceedings of the 2006 ACM SIGPLAN conference on Programming language design and implementation*, PLDI '06, pages 14–25, New York, NY, USA, 2006. ACM.

[39] Anant Agarwal Henry Hoffmann, David Wentzlaff. Remote Store Programming: Mechanisms and Performance. Technical Report MIT-CSAIL-TR-2009-017, MIT, 2009.

[40] Maurice Herlihy. Wait-free synchronization. *ACM Trans. Program. Lang. Syst.*, 13(1):124–149, 1991.

[41] Maurice Herlihy and J. Eliot B. Moss. Transactional memory: architectural support for lock-free data structures. In *Proceedings of the 20th annual international symposium on computer architecture*, ISCA '93, pages 289–300, New York, NY, USA, 1993. ACM.

[42] Carl Hewitt, Peter Bishop, and Richard Steiger. A universal modular ACTOR formalism for artificial intelligence. In *Proceedings of the 3rd international joint conference on Artificial intelligence*, IJCAI'73, pages 235–245, San Francisco, CA, USA, 1973. Morgan Kaufmann Publishers Inc.

[43] Holger H. Hoos and Thomas Sttzle. SATLIB: An Online Resource for Research on SAT. pages 283–292. IOS Press, 2000.

[44] J. Howard, S. Dighe, Y. Hoskote, S. Vangal, D. Finan, G. Ruhl, D. Jenkins, H. Wilson, N. Borkar, G. Schrom, F. Pailet, S. Jain, T. Jacob, S. Yada, S. Marella, P. Salihundam, V. Erraguntla, M. Konow, M. Riepen, G. Droege, J. Lindemann, M. Gries, T. Apel, K. Henriss, T. Lund Larsen, S. Steibl, S. Borkar, V. De, R. Van Der Wijngaart, and T. Mattson. A 48-Core IA-32 message-passing processor with DVFS in 45nm CMOS. In *Solid-State Circuits Conference Digest of Technical Papers (ISSCC), 2010 IEEE International*, pages 108 –109, Feburary 2010.

[45] Herbert H. J. HUM and James R. GOODMAN. Forward state for use in cache coherency in a multiprocessor system. Patent, 07 2005. US 6922756.

[46] Intel Corporation. An Introduction to the Intel QuickPath Interconnect, 2009.

[47] Intel Corporation. *Intel® Threading Building Blocks Reference Manual.* Number 315415-016. Jan 2012.

[48] Intel Corporation. *Intel® 64 and IA-32 Architectures Software Developer's Manual.* Number 325462-043US. May 2012.

[49] Intel Corporation. *Intel® Architecture Instruction Set Extensions Programming Reference.* Number 319433-012A. February 2012.

[50] Intel Labs. *SCC External Architecture Specification (EAS) Revision 0.934.*

[51] John D. Owens AND David Luebke AND Naga Govindaraju AND Mark Harris AND Jens Krger AND Aaron Lefohn AND Timothy J. Purcell. A Survey of General-Purpose Computation on Graphics Hardware. *Computer Graphics Forum*, 26(1):80–113, 2007.

[52] Alain Kägi, Doug Burger, and James R. Goodman. Efficient synchronization: let them eat QOLB. In *Proceedings of the 24th annual international symposium on Computer architecture*, ISCA '97, pages 170–180, New York, NY, USA, 1997. ACM.

[53] J. A. Kahle, M. N. Day, H. P. Hofstee, C. R. Johns, T. R. Maeurer, and D. Shippy. Introduction to the Cell multiprocessor. *IBM Journal of Research and Development*, 49(4.5):589 –604, July 2005.

[54] M. Kistler, M. Perrone, and F. Petrini. Cell Multiprocessor Communication Network: Built for Speed. *Micro, IEEE*, 26(3):10 –23, May 2006.

[55] Pranay Koka, Michael O. McCracken, Herb Schwetman, Xuezhe Zheng, Ron Ho, and Ashok V. Krishnamoorthy. Silicon-photonic network architectures for scalable, power-efficient multi-chip systems. *SIGARCH Comput. Archit. News*, 38(3):117–128, June 2010.

[56] P. Kongetira, K. Aingaran, and K. Olukotun. Niagara: a 32-way multithreaded Sparc processor. *Micro, IEEE*, 25(2):21 – 29, March 2005.

[57] Sanjeev Kumar, Christopher J. Hughes, and Anthony Nguyen. Carbon: architectural support for fine-grained parallelism on chip multiprocessors. In *Proceedings of the 34th annual international symposium on Computer architecture*, ISCA '07, pages 162–173, New York, NY, USA, 2007. ACM.

[58] Leslie Lamport. Specifying Concurrent Program Modules. *ACM Trans. Program. Lang. Syst.*, 5(2):190–222, 1983.

[59] E.-H. Lee, S. G. Lee, B. H. O, S. G. Park, and K. H. Kim. Fabrication of a hybrid electrical-optical printed circuit board (EO-PCB) by lamination of an optical printed circuit board (O-PCB) and an electrical printed circuit board (E-PCB). In A. M. Earman and R. T. Chen, editors, *Society of Photo-Optical Instrumentation Engineers (SPIE) Conference Series*, volume 6126 of *Society of Photo-Optical Instrumentation Engineers (SPIE) Conference Series*, pages 215–224, 2006.

[60] P.P.C. Lee, Tian Bu, and G. Chandranmenon. A lock-free, cache-efficient multi-core synchronization mechanism for line-rate network traffic monitoring. In *Parallel Distributed Processing (IPDPS), 2010 IEEE International Symposium on*, pages 1 –12, April 2010.

[61] P.P.C. Lee, Tian Bu, and G. Chandranmenon. A lock-free, cache-efficient multi-core synchronization mechanism for line-rate network traffic monitoring. In *Parallel Distributed Processing (IPDPS), 2010 IEEE International Symposium on*, pages 1 –12, April 2010.

[62] Victor W. Lee, Changkyu Kim, Jatin Chhugani, Michael Deisher, Daehyun Kim, Anthony D. Nguyen, Nadathur Satish, Mikhail Smelyanskiy, Srinivas Chennupaty, Per Hammarlund, Ronak Singhal, and Pradeep Dubey. Debunking the 100X GPU vs. CPU myth: an evaluation of throughput computing on CPU and GPU. In *Proceedings of the 37th annual international symposium on Computer architecture*, ISCA '10, pages 451–460, New York, NY, USA, 2010. ACM.

[63] Walter Lee, Rajeev Barua, Matthew Frank, Devabhaktuni Srikrishna, Jonathan Babb, Vivek Sarkar, and Saman Amarasinghe. Space-time scheduling of instruction-level parallelism on a raw machine. In *Proceedings of the eighth international conference on Architectural support for programming languages and operating systems*, ASPLOS-VIII, pages 46–57, New York, NY, USA, 1998. ACM.

[64] D. Lenoski, J. Laudon, K. Gharachorloo, W.-D. Weber, A. Gupta, J. Hennessy, M. Horowitz, and M.S. Lam. The Stanford Dash multiprocessor. *Computer*, 25(3):63 –79, March 1992.

[65] Chuanpeng Li, Chen Ding, and Kai Shen. Quantifying the cost of context switch. In *Proceedings of the 2007 workshop on Experimental computer science*, ExpCS '07, New York, NY, USA, 2007. ACM.

[66] E. Lindholm, J. Nickolls, S. Oberman, and J. Montrym. NVIDIA Tesla: A Unified Graphics and Computing Architecture. *Micro, IEEE*, 28(2):39 –55, March 2008.

[67] C.C. Liu, I. Ganusov, M. Burtscher, and Sandip Tiwari. Bridging the processor-memory performance gap with 3D IC technology. *Design Test of Computers, IEEE*, 22(6):556 – 564, November 2005.

[68] Lucien M. Censier and Paul Feautrier. A New Solution to Coherence Problems in Multicache Systems. *IEEE Transactions on Computers*, 27:1112–1118, 1978.

[69] M.M.K. Martin, M.D. Hill, and D.A. Wood. Token Coherence: decoupling performance and correctness. In *Computer Architecture, 2003. Proceedings. 30th Annual International Symposium on*, pages 182 – 193, June 2003.

[70] Timothy G. Mattson, Michael Riepen, Thomas Lehnig, Paul Brett, Werner Haas, Patrick Kennedy, Jason Howard, Sriram Vangal, Nitin Borkar, Greg Ruhl, and Saurabh Dighe. The 48-core SCC Processor: the Programmer's View. In *Proceedings of the 2010 ACM/IEEE International Conference for High Performance Computing,*

Networking, Storage and Analysis, SC '10, pages 1–11, Washington, DC, USA, 2010. IEEE Computer Society.

[71] D. May. The XMOS XS1 Architecture, 2009.

[72] David May. OCCAM. *SIGPLAN Not.*, 18(4):69–79, 1983.

[73] John M. Mellor Crummey and Michael L. Scott. Algorithms for scalable synchronization on shared-memory multiprocessors. *ACM Trans. Comput. Syst.*, 9(1):21–65, 1991.

[74] Inc MIPS Technologies. *MIPS64 Architecture for Programmers Volume 1: Introduction to MIPS64.* March 2011.

[75] David Mizell and Kristyn Maschhoff. Early experiences with large-scale Cray XMT systems. In *Proceedings of the 2009 IEEE international symposium on parallel & distributed processing*, IPDPS '09, pages 1–9, Washington, DC, USA, 2009. IEEE Computer Society.

[76] Simon Moore and Daniel Greenfield. The next resource war: computation vs. communication. In *Proceedings of the 2008 international workshop on System level interconnect prediction*, SLIP '08, pages 81–86, New York, NY, USA, 2008. ACM.

[77] S.W. Moore. *Multithreaded Processor Design.* Kluwer international series in engineering and computer science. Kluwer Academic Publishers, 1996.

[78] R. Nikhil. Bluespec System Verilog: efficient, correct RTL from high level specifications. In *Formal Methods and Models for Co-Design, 2004. MEMOCODE '04. Proceedings. Second ACM and IEEE International Conference on*, pages 69 – 70, June 2004.

[79] B. Nitzberg and V. Lo. Distributed shared memory: a survey of issues and algorithms. *Computer*, 24(8):52 –60, August 1991.

[80] Michael D. Noakes, Deborah A. Wallach, and William J. Dally. The J-machine multicomputer: an architectural evaluation. In *Proceedings of the 20th annual international symposium on computer architecture*, ISCA '93, pages 224–235, New York, NY, USA, 1993. ACM.

[81] M. Ohara, H. Inoue, Y. Sohda, H. Komatsu, and T. Nakatani. MPI microtask for programming the Cell Broadband Engine processor. *IBM Systems Journal*, 45(1):85 –102, 2006.

[82] Scott Owens, Susmit Sarkar, and Peter Sewell. A Better x86 Memory Model: x86-TSO. In *Proceedings of the 22nd International Conference on Theorem Proving in Higher Order Logics*, TPHOLs '09, pages 391–407, Berlin, Heidelberg, 2009. Springer-Verlag.

[83] Mark S. Papamarcos and Janak H. Patel. A low-overhead coherence solution for multiprocessors with private cache memories. In *Proceedings of the 11th annual international symposium on Computer architecture*, ISCA '84, pages 348–354, New York, NY, USA, 1984. ACM.

[84] Paul E. Mckenney. Memory Barriers: a Hardware View for Software Hackers, 2009.

[85] Juergen Ributzka, Yuhei Hayashi, Joseph B. Manzano, and Guang R. Gao. The elephant and the mice: the role of non-strict fine-grain synchronization for modern many-core architectures. In *Proceedings of the international conference on Supercomputing*, ICS '11, pages 338–347, New York, NY, USA, 2011. ACM.

[86] Ron Ho and Mark A. Horowitz. On-Chip Wires: Scaling and Efficiency, 2003.

[87] Daniel Sanchez, Richard M. Yoo, and Christos Kozyrakis. Flexible architectural support for fine-grain scheduling. In *Proceedings of the fifteenth edition of ASPLOS on Architectural support for programming languages and operating systems*, ASPLOS '10, pages 311–322, New York, NY, USA, 2010. ACM.

[88] Sanghoon Lee, D. Tiwari, Y. Solihin, and J. Tuck. HAQu: Hardware-accelerated queueing for fine-grained threading on a chip multiprocessor. In *High Performance Computer Architecture (HPCA), 2011 IEEE 17th International Symposium on*, pages 99 –110, February 2011.

[89] O. Serres, A. Anbar, S. Merchant, and T. El Ghazawi. Experiences with UPC on TILE-64 processor. In *Aerospace Conference, 2011 IEEE*, pages 1 –9, March 2011.

[90] M. Shah, J. Barren, J. Brooks, R. Golla, G. Grohoski, N. Gura, R. Hetherington, P. Jordan, M. Luttrell, C. Olson, B. Sana, D. Sheahan, L. Spracklen, and A. Wynn. UltraSPARC T2: A highly-treaded, power-efficient, SPARC SOC. In *Solid-State Circuits Conference, 2007. ASSCC '07. IEEE Asian*, pages 22 –25, November 2007.

[91] Sheng Li, S. Kuntz, J.B. Brockman, and P.M. Kogge. Lightweight Chip Multi-Threading (LCMT): Maximizing Fine-Grained Parallelism On-Chip. *Parallel and Distributed Systems, IEEE Transactions on*, 22(7):1178 –1191, July 2011.

[92] Mark Silberstein, Assaf Schuster, Dan Geiger, Anjul Patney, and John D. Owens. Efficient computation of sum-products on GPUs through software-managed cache. In *Proceedings of the 22nd annual international conference on Supercomputing*, ICS '08, pages 309–318, New York, NY, USA, 2008. ACM.

[93] Burton J Smith. Architecture and applications of the HEP multiprocessor computer system. *Real time signal processing IV*, 298:241248, 1981.

[94] A. Solomatnikov. *Polymorphic chip multiprocessor architecture.* ProQuest, 2009.

[95] P. Sweazey and A. J. Smith. A class of compatible cache consistency protocols and their support by the IEEE futurebus. *SIGARCH Comput. Archit. News*, 14(2):414–423, May 1986.

[96] Michael Bedford Taylor, Jason Kim, Jason Miller, David Wentzlaff, Fae Ghodrat, Ben Greenwald, Henry Hoffman, Paul Johnson, Jae-Wook Lee, Walter Lee, Albert Ma, Arvind Saraf, Mark Seneski, Nathan Shnidman, Volker Strumpen, Matt Frank, Saman Amarasinghe, and Anant Agarwal. The Raw Microprocessor: A Computational Fabric for Software Circuits and General-Purpose Programs. *IEEE Micro*, 22(2):25–35, 2002.

[97] The UPC Consortium. *UPC Language Specifications v1.2.* 2005.

[98] Tilera Corporation. Tile Processor User Architecture Manual, May 2011.

[99] J. Tolke and M. Krafczyk. TeraFLOP computing on a desktop PC with GPUs for 3D CFD. *Int. J. Comput. Fluid Dyn.*, 22(7):443–456, August 2008.

[100] D.M. Tullsen, J.L. Lo, S.J. Eggers, and H.M. Levy. Supporting fine-grained synchronization on a simultaneous multithreading processor. In *High-Performance Computer Architecture, 1999. Proceedings. Fifth International Symposium On*, pages 54 –58, January 1999.

[101] Wen-Hsiang Hu, Chifeng Wang, and N. Bagherzadeh. Design and Analysis of a Mesh-based Wireless Network-on-Chip. In *Parallel, Distributed and Network-Based Processing (PDP), 2012 20th Euromicro International Conference on*, pages 483 –490, February 2012.

[102] David Wentzlaff, Patrick Griffin, Henry Hoffmann, Liewei Bao, Bruce Edwards, Carl Ramey, Matthew Mattina, Chyi-Chang Miao, John F. Brown III, and Anant Agarwal. On-Chip Interconnection Architecture of the Tile Processor. *IEEE Micro*, 27(5):15–31, 2007.

[103] Colin Whitby Strevens. The transputer. *SIGARCH Comput. Archit. News*, 13(3):292–300, June 1985.

[104] Peng Wu, Maged M. Michael, Christoph von Praun, Takuya Nakaike, Rajesh Bordawekar, Harold W. Cain, Calin Cascaval, Siddhartha Chatterjee, Stefanie Chiras, Rui Hou, Mark Mergen, Xiaowei Shen, Michael F. Spear, Hua Yong Wang, and Kun Wang. Compiler and runtime techniques for software transactional memory optimization. *Concurr. Comput. : Pract. Exper.*, 21(1):7–23, 2009.

[105] William A. Wulf and C. G. Bell. C.mmp: a multi-mini-processor. In *Proceedings of the December 5-7, 1972, fall joint computer conference, part II*, AFIPS '72 (Fall, part II), pages 765–777, New York, NY, USA, 1972. ACM.

[106] M. Yuffe, E. Knoll, M. Mehalel, J. Shor, and T. Kurts. A fully integrated multi-CPU, GPU and memory controller 32nm processor. In *Solid-State Circuits Conference Digest of Technical Papers (ISSCC), 2011 IEEE International*, pages 264 –266, Feburary 2011.

[107] Weirong Zhu, Vugranam C Sreedhar, Ziang Hu, and Guang R. Gao. Synchronization state buffer: supporting efficient fine-grain synchronization on many-core architectures. In *Proceedings of the 34th annual international symposium on Computer architecture*, ISCA '07, pages 35–45, New York, NY, USA, 2007. ACM.