A CHIP MULTI-CLUSTER ARCHITECTURE WITH LOCALITY AWARE TASK DISTRIBUTION

A thesis submitted to the University of Manchester
for the degree of Doctor of Philosophy
in the Faculty of Engineering and Physical Sciences

2007

Matthew James Horsnell
School of Computer Science

Contents

List of Tables

List of Figures

List of Algorithms

Abstract

Chip MultiProcessor (CMP) architectures are fast becoming ubiquitous. Their widespread adoption has been motivated by three dominant factors; power and thermal limits have constrained higher clock frequencies, the memory wall has expedited concurrency as a means of maintaining performance, and technology advances have increased transistor budgets enabling the integration of multiple cores on a single chip. It is anticipated that a trend of increasing the number of cores with increasing transistor budgets will emerge, and that within the next decade it will be feasible to integrate up to 128 cores within a single chip architecture.

This thesis investigates the scaling limitations of current single bus CMP architectures and proposes a Chip Multi-Cluster (CMC) architecture as a feasible approach for future many-core designs. A novel cache coherence protocol and hardware support for maintaining coherence across multiple clusters is presented. Additionally, support at the hardware/software interface is provided to allow locality-aware thread creation and distribution in order to best utilise the architecture. Several possible implementations of the CMC architecture are studied through cycle accurate simulation using multithreaded benchmarks.

Declaration

No portion of the work referred to in this thesis has been submitted in support of an application for another degree or qualification of this or any other university or other institution of learning.

Copyright

Acknowledgements

I would like to thank my supervisor Professor Ian Watson for all of the help, encouragement, support and constructive feedback he has given me over the last four years. I would also like to thank Ian Rogers, Ahmed El-Mahdy and Andrew Dinn for proof reading this thesis and for the many discussions that have continued to motivate me.

I would like to thank all of the members of the Jamaica project and the APT group. Those with whom I have been able to discuss research ideas and those who have provided lively conversations over a few pints. I would also like to thank my friends outside of computer science for keeping me sane by taking me away from computers and remaining in contact with me throughout.

Special thanks to my parents for their continued support and encouragement, and to my brother Jonathan and sister Katherine for their enduring friendship.

Finally, and certainly not least, a debt of gratitude is owed to Laura for her unfaltering patience, understanding and love.

CHAPTER 1

Introduction

A chip multi-cluster architecture based on a novel multi-level cache coherence protocol is presented in this thesis. This architecture supports interleaved multi-threading and provides facilities for lightweight locality aware thread distribution. The architecture is simulated using a novel cycle-level simulation platform and is used to evaluate the concepts associated with increasing the number of cores, bus contention and wire delay, scaling the memory hierarchy, and locality aware task distribution.

1.1 Motivation

Over 40 years ago Gordon Moore observed that the total number of devices integrated on a chip doubled every 12 months [111]. Based on this observation he boldly predicted that this trend would continue throughout the 1970s and would subsequently slow down to a doubling every 24 months in the 1980s. This prediction, commonly referred to as Moore's Law, triggered a revolution in microarchitecture innovation and design that has delivered enormous increases in computing power.

During the last decade alone, the number of transistors integrated on a single die has doubled every 24 months and the relative performance of microprocessors when executing SPECint benchmarks has grown by over 75 times [63]. In the same period the type of applications processed by microprocessors has diversified enormously. With further expansion of broadband internet, multimedia, gaming and mobile communication the complexity of such application areas continue to push a demand for yet more performance increases.

Sustaining this performance growth has, to date, largely been achieved through technology scaling. In the last 10 years mainstream semiconductor technology has scaled feature sizes down from 350nm to 65nm, enabling operating frequencies to increase from 200MHz to 3.2GHz, on the Intel Pentium Pro and Pentium 4 respectively. This frequency increase, of 16 times, has provided the majority of the 75 times performance increase, with the remaining increase due to microarchitecture innovation and exploiting higher transistor budgets. Both frequency increases and exploiting higher transistor budgets are becoming increasingly difficult in single processor designs.

Until recently increasing transistor budgets have been exploited by increasing the pipeline depth, issue width and reorder buffers of single processor superscalar architectures. Unfortunately, the performance gained in this manner has been diminishing [20] and further small performance gains require discouragingly complex additional hardware. A recent study [53] showed that there is a growing discrepancy between the increase in area employed by a new microarchitecture and the increase in performance, with the increase in performance growing at the square root of the increase in area. A productivity gap is also emerging, as designers and design tools are not able to keep pace with the increase in complexity of modern designs. This was highlighted by the International Technology Roadmap for Semiconductors as a grand challenge [139].

As mentioned previously, the rapid increase in clock frequency, 40 percent per year for the past 15 years, has been the dominant factor in microprocessor performance increases. This speed increase has come from two sources: smaller, faster transistors and deeper pipelines with shorter critical paths. For two reasons this increase has diminished in the past few years. Firstly, the rapid increases in speed have hastened the emergence of a *power wall*. Simply put, the power consumed by modern microprocessors is becoming too costly for the end user, and more

importantly perhaps, the heat dissipated is becoming too expensive to cool [38]. Secondly, as feature size decreases transistor switching speed increases, however wires are not scaling as quickly [17]. This is leading to wire delay limited circuits, where the percentage of the chip accessible in one clock cycle is decreasing per generation.

With the main techniques responsible for increases in microprocessor performance rapidly expiring, a shift towards different design strategies capable of maintaining performance increases is currently underway. Although many alternatives were initially proposed in a special issue of IEEE Computer [19], parallel architectures, in the form of Chip Multiprocessors (CMP) have become the focus of most major microprocessors roadmaps [81, 127, 78, 108]. These architectures are able to overcome or avoid design challenges of modern microprocessors and at the same time continue increasing performance, largely by exploiting parallelism.

The remainder of this chapter introduces and expands on some of the key issues facing modern microarchitecture design. This is followed by a discussion of how computer architects are innovating in order to overcome these challenges.

1.2 Microprocessor Design Challenges

As mentioned previously, the application space for general purpose microprocessors is vast and is growing as new technologies and application areas are discovered. Applications, whether multimedia, gaming, communications or scientific, require microprocessors that are capable of processing a variety of tasks, many within imposed timing constraints. At the same time computer systems are used to run multiple applications concurrently, adding additional complexity to the workload of a general purpose microprocessor.

For these purposes it is desirable that each successive generation of microprocessor is able to face a changing and growing application space, and is able to remain capable of processing workloads into the future. At the same time the end user of a computer system does not want the cost of ownership to become overly expensive, so power consumption is a key concern. The design of processors optimised for performance, power and cost requires a carefully balanced architecture which

addresses numerous challenges. This section describes the major challenges that architects face in designing modern architectures.

1.2.1 Wire Delay

As semiconductor technology advances, transistors are becoming smaller and more transistors can be integrated onto a single silicon die. These transistors consume less power and the time taken by them to switch state, the *gate delay*, decreases. In real terms transistors, and hence logic, on a chip are becoming cheaper and faster.

In order to connect transistors as their feature sizes decrease, the width of wires writhin a given technology must also decrease. This reduces the cross-sectional area of the wires (A), which increases the resistance (R) per unit of length (L) because resistance is inversely proportional to area, Equation 1.1. In the equation ρ is the resistivity constant of the material.

$$R = \frac{\rho L}{A} \tag{1.1}$$

$$\tau = RC \tag{1.2}$$

This means that the delay (τ) through a wire, Equation 1.2, where C is wire capacitance, will only decrease linearly with its length, which depends on the underlying transistor technology [17]. As wire delay is only decreasing linearly and gate delay is decreasing more rapidly, circuits are increasingly becoming wire delay limited. This happens because the number of gates reachable in a single clock cycle is decreasing.

The overall impact that this has on a microarchitecture is that shrinking a wire delay limited circuit will not make it run any faster, as less of the silicon is available within one clock cycle [66]. In fact, it was estimated that when the semiconductor technology gets down to a 0.1μm process, only 16% of the die will be reachable within a single clock cycle [104], Figure 1.1.

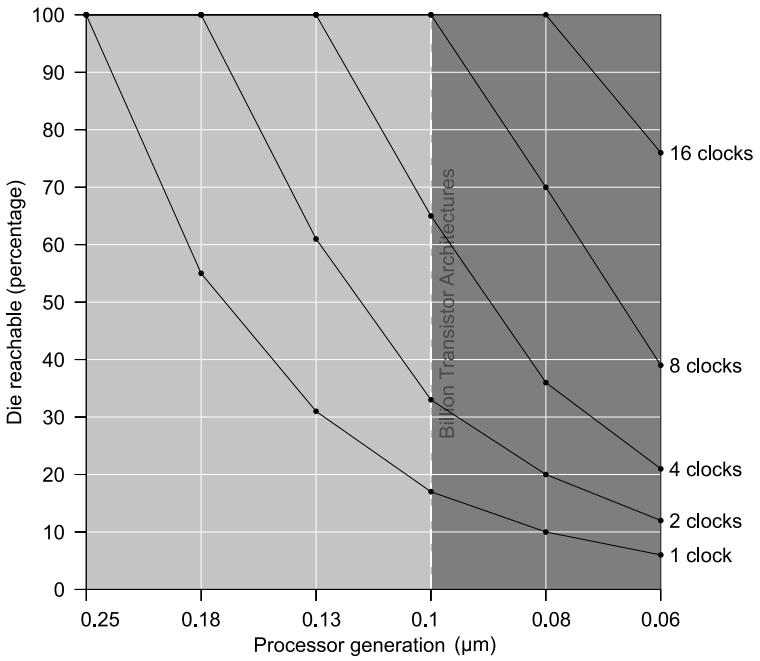


Figure 1.1: *Predicted percent of die reachable by each generation [104].*

1.2.2 Memory Gap

In order for a microprocessor to operate efficiently it needs to be fed with a steady stream of instructions to process and to have access to the data required by those instructions. In today's architectures access to main memory, to load or store instructions and data, is often a bottleneck. This bottleneck is caused by the discrepancy in memory and processor speeds. While processor clock speeds have increased by approximately 55% per year since 1980, memory performance has only been increasing at less than 10%, creating a widening memory gap [62], see Figure 1.2.

This increasing memory gap reduces the benefit of increased operating frequencies; whenever a processor requires access to a piece of data or instruction not currently in a local cache it has to wait. Access to memory, in modern microprocessors, takes hundreds of cycles during which time the processor remains idle. This gap is especially evident in commercial applications that are transaction intensive.

In order to alleviate this problem, smaller *cache* memories are used to keep frequently accessed data local to the processor and most modern architectures now

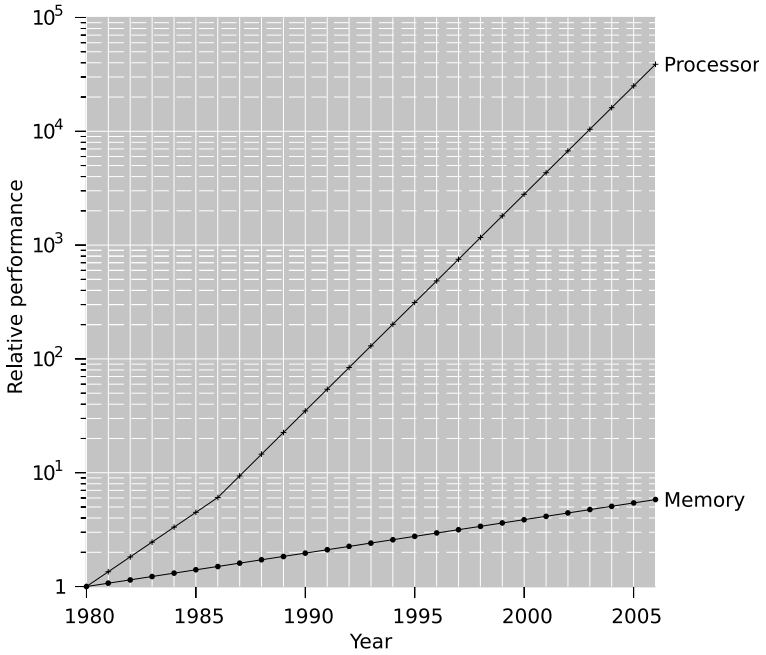


Figure 1.2: *A growing gap between memory speed and processor speed leads to bottlenecks during data intensive workloads [62].*

contain at least two levels of cache memory, of which many different configurations have been researched [75, 42]. Complimentary to storing frequently used data are schemes for fetching data in advance of requirement. These *prefetching* schemes have been attempted both in software [23, 113] and hardware [28, 162] to reduce the impact of the memory wall further.

1.2.3 Limits of Instruction Level Parallelism

Many modern processors contain hardware support for exploiting instruction level parallelism (ILP). ILP exists where multiple instructions are independent from each other and can be executed simultaneously. Independent instructions found within an instruction stream can be issued to multiple functional units within the processor for execution, and modern superscalar architectures are capable of dispatching four or more instructions per clock cycle to separate functional units.

Whilst this approach has enabled performance increases, and fundamentally allows more than one instruction to be executed in each clock cycle, it does have limitations. In particular instructions within the same thread usually display a

high degree of dependence, and finding further independent instructions requires the ability to look further ahead in the instruction stream, which requires additional hardware.

A plethora of studies have looked at the limits of ILP [10, 22, 166, 129]. Many have shown that even in the presence of theoretical perfect caches[1], and perfect branch prediction, the maximum attainable ILP is still only in the order of 10 to 100 instructions per cycle (IPC) [62]. In practice ILP very rarely attains greater than 4 IPC with relatively complex but manageable hardware. Beyond this, the complexity of large instruction fetch windows, broadcast networks which suffer from wire delays, multiple functional units and centralised control becomes impractical. Some of this hardware, for example register bypass logic, grows quadratically [123] when attempting to exploit further ILP and performance gains diminish.

1.2.4 Power Limits

Power consumption has gone from being a factor that needed to be considered when designing a new architecture, to becoming a first order constraint on the design of new architectures. The limit to acceptable power consumption is usually realised when the ability to dissipate the heat from a processor becomes difficult. With the recent rapid increases in clock frequency and the continual increase in chip transistor density, the heat dissipated by modern high end microprocessors is becoming unmanageable.

Power consumption in a processor comprises a static component, called leakage power, and a dynamic component, called switching power, Equation 1.3. Traditionally the static component has been a fraction of the dynamic component, however in modern semiconductor technologies this is changing.

$$\begin{aligned} Power_{total} &= Power_{static} + Power_{dynamic} \\ &= V \times I_{leakage} + \frac{CV^2 f_{clock}}{2} \end{aligned} \tag{1.3}$$

[1] A theoretical perfect cache would have the property that each access would result in a cache hit.

Process	Old Architecture	Area mm^2	New Architecture	Area mm^2	Area Increase
1.0μm	i386 (compaction)	42.25	i486 (lead)	132.25	3.1
0.7μm	i486 (compaction)	90.25	Pentium (lead)	289	3.2
0.5μm	Pentium (compaction)	148.84	Pentium Pro (lead)	299.29	2.0

Table 1.1: *Growth in area between successive generations of Intel architectures.*

As the feature size decreases with semiconductor technology, the size of the transistors and hence their capacitance (C) decreases. This reduced capacitance decreases the transistor switch time, or gate delay, leading to increased performance. However as the feature size decreases the voltage (V) must be lowered to reduce the interference between the closer components and in order to meet thermal requirements for the design.

In order to keep the chip functioning correctly at a reduced operating voltage the *threshold voltage*, the threshold at which the transistor switches state, must be decreased. A lower threshold voltage brings it closer to ground which increases the static leakage current, $I_{leakage}$, increasing the static power consumption. Recent research has focused on techniques to reduce static power consumption [115].

1.2.5 Design Complexity

Each semiconductor technology generation is allowing more transistors to be integrated onto a single silicon chip, and potentially these additional transistors can be incorporated into future microarchitecture designs. Recently Intel released the Montecito processor [107], a dual-core, dual-threaded Itanium architecture, which incorporated 1.72 billion transistors in the design, taking silicon chips into the era of billion transistor architectures.

Unfortunately, growth in performance in a new microarchitecture is declining in subsequent generations and is approximately proportional to the square root of the growth in area of the microarchitecture in any given technology generation. Taking the x86 family of architectures as an example, the growth in area of approximately 2-3 times, Table 1.1, was accompanied by only a 1.5 - 1.7 times increase in performance.

This discrepancy is due to the hardware that was added in subsequent generations of microarchitecture. As previously mentioned, hardware for exploiting ILP

suffers from diminishing returns, as does adding larger cache memories. At some point quadratic increases in the size of the hardware only achieve linear increases in performance.

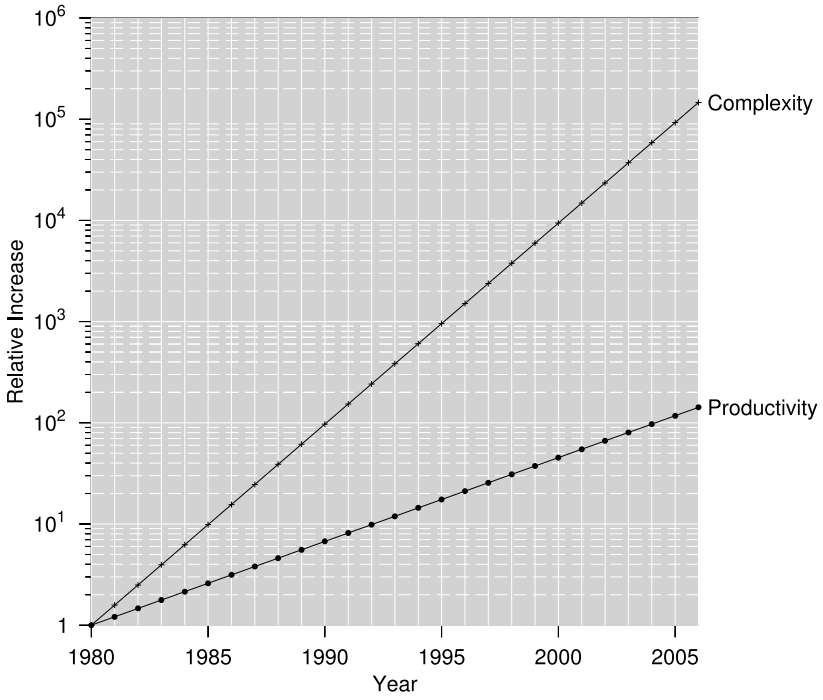


Figure 1.3: *A growing gap between increases in the complexity of a chip and productivity of design engineers and tools.*

Another important issue associated with design complexity is designer productivity. The International Technology Roadmap for the Semiconductor Industry (ITRS) 2005 [139] continued to highlight the gap between design complexity, measured as the total number of transistors on a chip, and designer productivity measured as transistors designed-in per staff member per month, Figure 1.3. The number of transistors on-chip is growing at a rate of 58% per year, but design productivity is only increasing at a rate of 21% per year. In order to close the productivity gap the ITRS stated that reuse, testing and verification must all improve by over 2 times.

1.3 Design Solutions

Computer architects have responded, with many novel solutions, to the current problems facing microprocessor design. In the following section some of these solutions are presented.

1.3.1 Exploiting Parallelism

As increases in the underlying circuit speed decline, due to issues such as power density and clock distribution, it appears that future performance will need to come from doing more work in parallel. As mentioned previously, Section 1.2.3, architectural techniques capable of extracting ILP are now reaching practical limits, motivating the need to look elsewhere for parallelism.

Just over 40 years ago, a taxonomy was proposed placing all computer architectures into four categories based on the parallelism in both the instruction and data streams [48]. This model is still useful for explaining where additional parallelism can be found in future microprocessors.

Single Instruction, Single Data Stream (SISD)

This category includes the uniprocessor, where a single instruction stream is processed against a single data stream. As discussed parallelism in SISD architectures can be extracted using ILP techniques, as is the case in superscalar architectures [123]. In addition to the ILP extracted through complex hardware, simpler hardware in combination with complex compiler techniques can be used to define explicitly parallelism in a single instruction stream, as is the case in Very Long Instruction Word (VLIW) architectures [45]. Both of these techniques have reached practical limits and further advances require discouraging expense in hardware with low utilisation.

Single Instruction, Multiple Data Stream (SIMD)

In SIMD architectures a single instruction is executed by multiple Processing Elements (PEs) on different data streams. SIMD architectures exploit *data level*

parallelism (DLP), by applying the same operation to multiple data items in parallel. Each PE has its own data memory, hence multiple data streams, however each PE is driven by the same instruction streams, usually from a single control processor responsible for fetching and dispatching each instruction.

For multimedia and scientific applications, which exhibit significant amounts of DLP, this approach is very efficient. As these applications have rapidly migrated into the desktop space, architectures have incorporated SIMD extensions into the instruction set [126, 39, 118] and provided special purpose hardware for executing these instructions. SIMD instructions are now ubiquitous in modern general purpose architectures, accelerating cryptographic, media encoding and decoding, and graphics processing.

The performance of SIMD architectures is limited only by the amount of DLP available in any given application, as the addition of further PEs is relatively cheap, in comparison to the structures associated with extending superscalar techniques.

Multiple Instruction, Single Data Stream (MISD)

MISD architectures process a single data memory with multiple instruction streams. An implementation of a MISD architecture [58] has been shown to be useful for very specific tasks, such as fast pattern matching in large data streams for which there is no efficient index, and hence multiple different search tasks can be executed on the same data in parallel. It appears that MISD architectures have not shown significant benefits in any general purpose application areas, and as such no commercial architecture of this type has been developed.

Multiple Instruction, Multiple Data Stream (MIMD)

In MIMD architectures multiple processors execute independent instruction streams on independent data streams. Each processor in a MIMD architecture executes a separate thread of control. That is, they execute independent instruction streams on largely independent[2] data streams in parallel. This *thread-level parallelism*

[2]In MIMD architectures shared memory coherence, in particular atomic primitives such as synchronisation, may prevent the streams from being fully independent.

(TLP) is far more flexible than DLP and it can be exploited by a larger set of the application space of general purpose microprocessors. For this reason TLP is perhaps the best candidate to achieve future performance gains in general purpose microprocessors.

Thread Level Parallelism

Many modern programming languages allow programmers to define, explicitly, independent threads of control within a program all of which can be executed in parallel. Additionally multi-processing operating systems allow multiple processes to be run concurrently. Each of these processes may be run in parallel, and in turn may contain threads that can also be run in parallel. Prior to MIMD architectures, SISD architectures relied on the operating system software to switch among concurrent processes and threads, ensuring that each was allowed sufficient execution time within the same processor to continue making forward progress, Figure 1.4 (a).

In order to accelerate TLP, many modern processors, especially in the server class, incorporate hardware that allows a single processor to maintain information for multiple threads, switching between each thread at a hardware level and filling empty slots in the processor's pipeline on long latency cache misses, Figure 1.4 (b). In highly threaded applications, of which the operating system is an example, multi-threaded hardware can help with hiding the long idle times associated with data and instruction accesses that miss local or intermediate caches. Due to the increasing memory gap, Section 1.2.2, these idle times can be several hundreds of clock cycles during which time, in the absence of multi-threading hardware, the processor would effectively remain idle. In some data intensive applications, such as database workloads, the idle time in modern microprocessors is as much as 75% [90]. A recent study [53], stated that adding multithreading hardware, in the study two threads, to an existing architecture adds approximately 10% additional logic to the CPU, increases the maximum power by less than 10% but can increase throughput by over 30%.

Another approach to exploiting TLP in hardware is to add additional processors to the architecture, via an off-chip interconnect at the board or multi-chip module (MCM) level, or, with billion transistor budgets, increasingly in the same

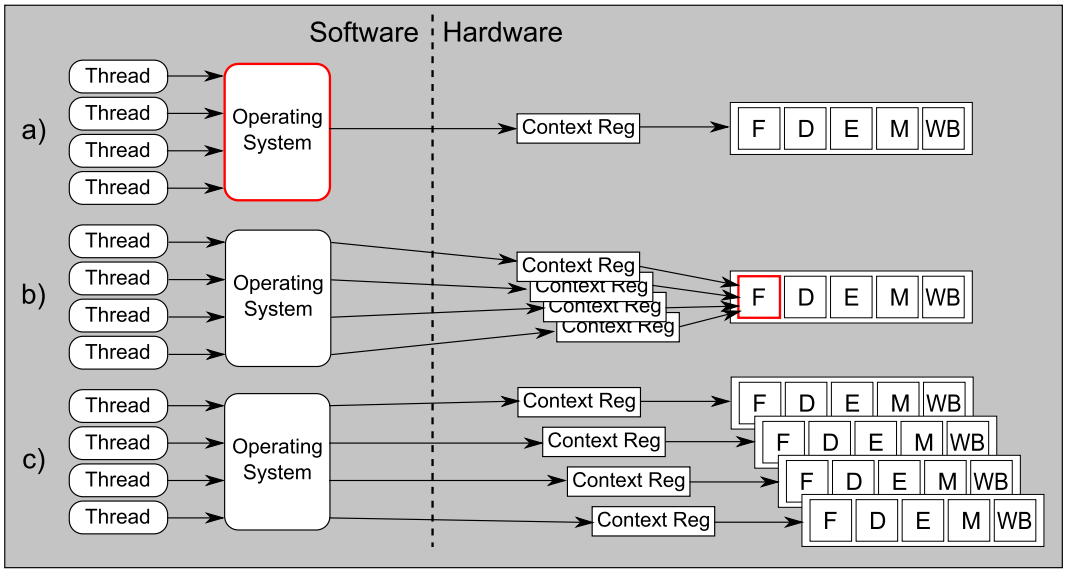


Figure 1.4: *Exploiting thread-level parallelism: a) software scheduling, b) multi-threaded hardware, c) multi-processor hardware.*

silicon die, Figure 1.4 (c). Additional processors can provide linear increases in performance with die size on transactional workloads such as TPC-C [34]. CMPs, where multiple processors are integrated on a single chip, can provide performance increases even on applications exhibiting fine-grained TLP.

1.3.2 Partitioned Designs

As mentioned in Section 1.2.1, wire delay is diminishing the area of a chip reachable in a single clock cycle with each generation of process technology. This motivates the need for designs where cross chip communication is minimised.

A new class of parallel architectures have been designed with wire delay treated as a first order design constraint. These communication-centric architectures such as RAW [165], Smart Memories [102] and TRIPS [137] keep the length of critical paths in the design to within 1 or 2 cycles. Whilst these architectures overcome limits due to wire delay they also impose a message-passing [165] or data-flow [141] paradigm of programming onto the compiler or software.

CMPs can also be designed to overcome wire delay limits. Each of the many small processing cores on CMPs take up a relatively small area on the total chip,

minimising the size of critical paths within each core. Only infrequently used and therefore less latency critical wires, connecting processors and caches, need to be long.

1.3.3 Bridging the Memory Gap

The memory gap, as mentioned in Section 1.2.2, is constantly increasing the penalty, in cycles, of memory loads and stores. In modern processors, such as the Intel Pentium 4, the cost of accessing main memory can be as large as 200-300 cycles. For this reason today's high performance processors employ multiple levels of cache memory to help reduce the performance gap.

Increasing the size of on-chip caches decreases the chance of a memory operation in the processor having to go all the way to main memory. In today's semiconductor technologies it is possible to integrate extremely large caches on-chip. Indeed, Intel's Montecito [107] processor included nearly 27MB of on chip cache memory, improving the performance by almost a factor of 2 over previous generations of the same architecture, Figure 1.5.

Cache memories cannot totally close the memory gap, as untouched data, when first loaded must always come from main memory unless prefetched in advance using hardware [28] or software [23]. It is however possible to improve performance even during loads and stores to untouched areas of memory by exploiting *memory-level parallelism* (MLP) [30]. MLP can be exploited by overlapping multiple accesses to main memory during the period that the processor is idle waiting for the first access to resolve.

To illustrate the potential of MLP as a performance booster, consider a memory-bound application that spends two-thirds of its execution time in off-chip accesses, doubling the MLP can halve the time spent in these accesses and potentially improve performance by 25%. As long latency memory accesses are fairly dynamic, occurring when a cache has evicted previously held data or has not yet seen it, hardware schemes are needed to look for memory accesses that can be overlapped.

One such scheme is *hardware scouting* [27]. When a processor is forced to stall on a memory operation, a scout thread is invoked. The scout thread's sole purpose is to run ahead in the instruction stream and look for memory accesses, while the real

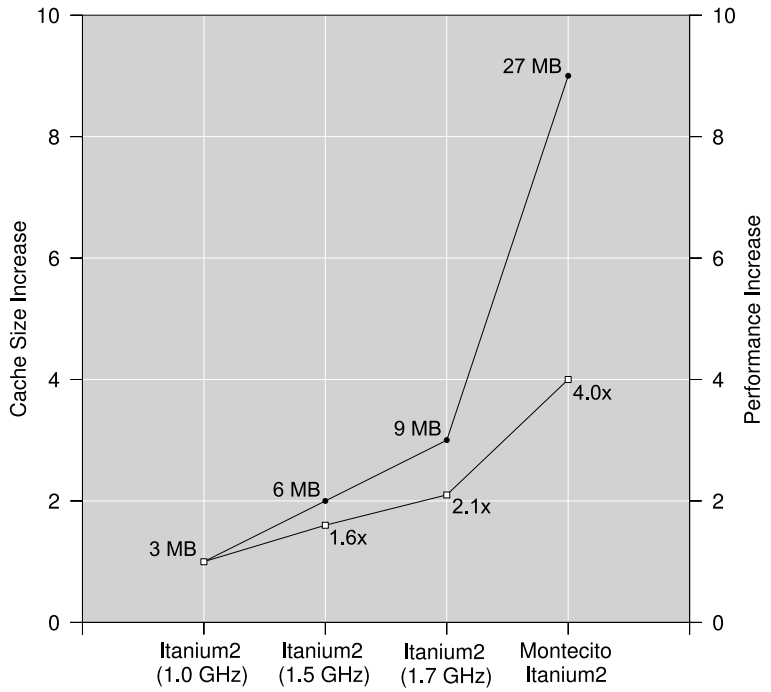


Figure 1.5: *The effect of increased cache size (MB) on performance (×) of the Itanium 2 processor [107].*

thread is stalled waiting for the initial access to come back. The hardware scout can pass control flow operations such as branches and jumps taking whichever path is deemed most likely, and can continue scouting for memory accesses many hundreds of cycles in advance of the real thread. Obviously when the initial memory access is resolved the real thread is switched back in, and continues from the point at which it stalled. Any memory accesses that were overlapped will have been fetched into local caches and the cost for the following accesses is reduced. Hardware scouting can be more efficient, in terms of logic area consumed, than simply increasing the size of caches, as illustrated in Figure 1.6.

1.3.4 Design Abstraction and Replication

Two methods by which the design complexity problem can be addressed are abstraction and replication. The International Technology Roadmap for Semiconductors [139], has continually outlined the need for both of these methods to be increased by at least a factor of two in order to bridge the productivity gap.

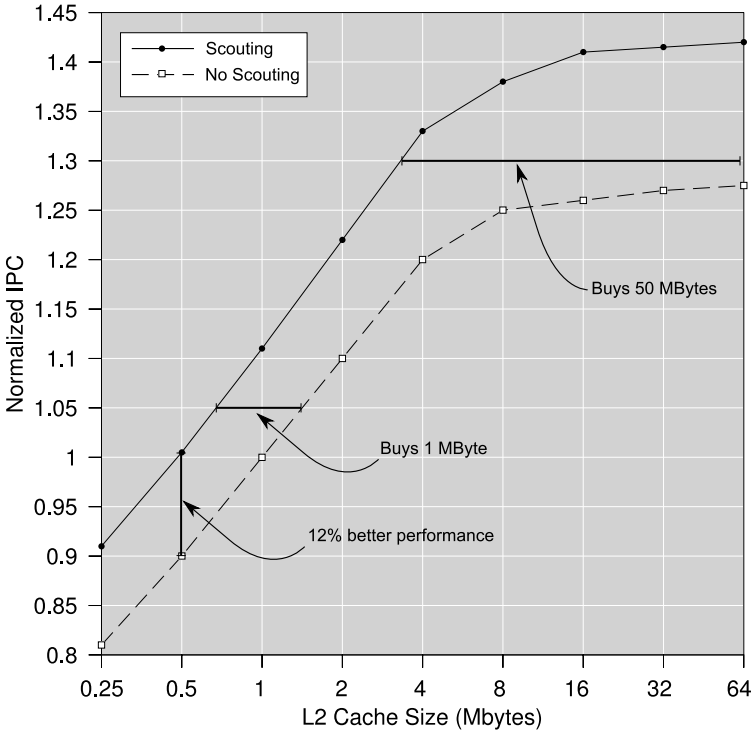


Figure 1.6: *Comparison of hardware scouting and increasing cache size [27].*

Abstraction can be addressed by designing systems at a higher level. Initially circuits were designed at the transistor level, then at the gate level using libraries, and more recently at the macro block level. With billion transistor budgets the abstraction level may have to be raised higher once more, possibly to the processor or multiple processor level. Next generation architectures can be composed of multiple copies of the previous generations in order to best utilise the increase in transistor space.

Replication within an architecture is also necessary. This can already be seen with current generation chip multiprocessors. Sun's Niagara processor [81] contains 8 identical Sparc in-order cores, IBM's Power 5 [78] and Intel's Core Duo [108] architectures both contain two identical superscalar processors, and STI's[3] Cell processor [127] contains one Power processor and eight synergistic processors. All of these architectures have reduced the design time normally associated with a new architecture through successful replication and reuse.

[3]STI - a collaboration by Sony, Toshiba and IBM

1.4 Summary

In this introduction, the major challenges facing modern microprocessor design have been outlined. These challenges and the demand for continual performance gains, have motivated research within this area. Moreover some of the solutions to these challenges have been discussed, and many come from a shift to parallelism as a design paradigm. In this context CMP architectures have been shown to have significant potential for future general purpose high performance processors.

1.5 Research Aims

The research presented in this thesis focuses on CMP architectures. An architecture is presented that scales beyond the current generation of CMPs by incorporating a multi-level cache hierarchy on a chip, allowing the notion of Chip Multiple-Cluster (CMC) architectures. In order to facilitate this clustering, a novel multi-level cache hierarchy is presented as well as a novel cache coherence protocol to maintain shared memory coherency. A scheme allowing locality based task distribution is presented, showing that in such architectures task isolation and task affinity can be used to improve performance.

1.6 Contributions

The contributions made by the work presented in this thesis are summarised as follows:

- A multi-level shared memory cache coherence protocol.
- Cache hardware to support a multi-level shared memory cache coherence protocol.
- An instruction set extension and mapping mechanism for exploiting cache locality between threads.

- A simulation platform capable of evaluating, through cycle-level simulation, chip multiprocessor and chip multi-cluster architectures containing upto and including 128 cores.
- A fully cache coherent study, using real multithreaded applications, of the effects and performance of large scale chip multiprocessor and chip multi-cluster architectures.

1.7 Thesis Structure

The rest of the thesis is structured as follows:

Chapter 2 reviews the availability of parallelism within software and the exploitation of parallelism within software and architecture design.

Chapter 3 outlines the Jamaica CMP which is the base architecture subsequently extended in later chapters. The chapter also presents the simulation platform used to investigate and analyse the architecture, and describes the software toolchain supporting it.

Chapter 4 introduces a multi-level cache coherence protocol capable of maintaining shared memory coherence across multiple on-chip clusters.

Chapter 5 presents the hardware extensions necessary to implement the coherence protocol. In particular a deadlock free queueing mechanism is described.

Chapter 6 introduces an extension to the instruction set which allows software to exploit locality by controlling the affinity of distributed threads.

Chapter 7 evaluates the performance of the multi-level coherence protocol, the architecture supporting it, and the optimisations using locality aware task distribution.

Chapter 8 concludes the thesis by summarising the contributions made and suggesting future directions of research that could be conducted.

Finally, the Appendix includes details of the Jamaica instruction set for reference.

1.8 Publications

Accepted Papers

- M. J. Horsnell, J. Zhao, I. Rogers, A. Dinn, C. K. Kirkham, I. Watson, *Optimizing Chip Multiprocessor Work Distribution using Dynamic Compilation*, European Conference on Parallel and Distributed Computing, Rennes, France, August 28-31, 2007. Lecture Notes in Computer Science, Volume 4641/2007, pages 258–267.

- I. Rogers, M. J. Horsnell, I. Watson, *Virtualization and chip multiprocessor memory management: the JAMAICA architecture*, 5^{th} UK Memory Management Network Workshop, University of Glasgow, UK, July 2005.

- M. J. Horsnell, I. Watson, *Harnessing Java for Novel Chip-Multiprocessor Architecture Simulations*, PREP2005 Conference, Lancaster, UK, March 2005.

Abstracts

- M. J. Horsnell, I. Watson, *Simulating the Jamaica CMP Architecture*, HiPEAC ACACES, L'Aquila, Italy, July 2005.

- M. J. Horsnell, I. Watson, *Cycle-Accurate, Distributed Chip Multiprocessor Simulation*, PREP2004 Conference, Hertfordshire, UK, March 2004.

CHAPTER 2

Application and Architectural Parallelism

With the gains available from conventional uniprocessor architecture techniques diminishing with each successive generation of processor technology, parallel computer architectures are being embraced by industry and researchers to provide scalable, consistent performance increases.

Parallel architectures consist of multiple processing elements that cooperate in order to solve problems, ideally in a shorter period of time than solving the problem using a single processing element alone. There exist three constraints on the performance attainable from a parallel architecture: the available parallelism in the application, the parallelism available in the hardware and the efficiency and nature of distributing and scheduling parallel work.

2.1 Application Parallelism

For a parallel computer architecture to realise any speed-up during the execution of a given workload, the workload must to some extent be amenable to parallelisation. The amount of an application that can be successfully parallelised and

executed concurrently on a parallel architecture determines the extent to which the parallel architecture is capable of reducing the overall execution time.

2.1.1 Amdahl's Law

The correlation between the amount of parallelism in a given code and the maximum speedup from parallel execution is commonly referred to as Ahmdahl's Law [4], and is a demonstration of the law of diminishing returns. The maximum theoretical speedup S, attainable from running an application on N processors, is shown in Equation 2.1, where P is the percentage of the code that can be parallelised.

$$S = \frac{1}{(1 - P) + \frac{P}{N}} \tag{2.1}$$

When Amdahl's law is applied, as shown in Figure 2.1, applications containing large percentages of parallel code sections do not guarantee large performance speedups. When $P = 0.9$, that is 90% of the code in a target application is amenable to parallelisation, and assuming no overheads associated with the parallel execution of the code, a 16 processor machine realises a speedup of less than 7. This result may initially appear disappointing, however, for the reasons outlined in Section 1.2, it may be infeasible to design and build a processor that can run at 7 times the clock frequency, whilst remaining inside a given power budget, therefore it may be more cost and time effective to replicate multiple processing cores from an existing design, in order to achieve the additional performance.

2.1.2 Implicit and Explicit Parallelism

With the amount of parallelism available in an application determing the amount of speedup attainable on a parallel architecture, it becomes essential to locate or introduce parallelism into an application. Locating parallelism within existing code is referred to as exploiting *implicit* parallelism, introducing or programming-in parallel sections of code is referred to as exploiting *explicit* parallelism.

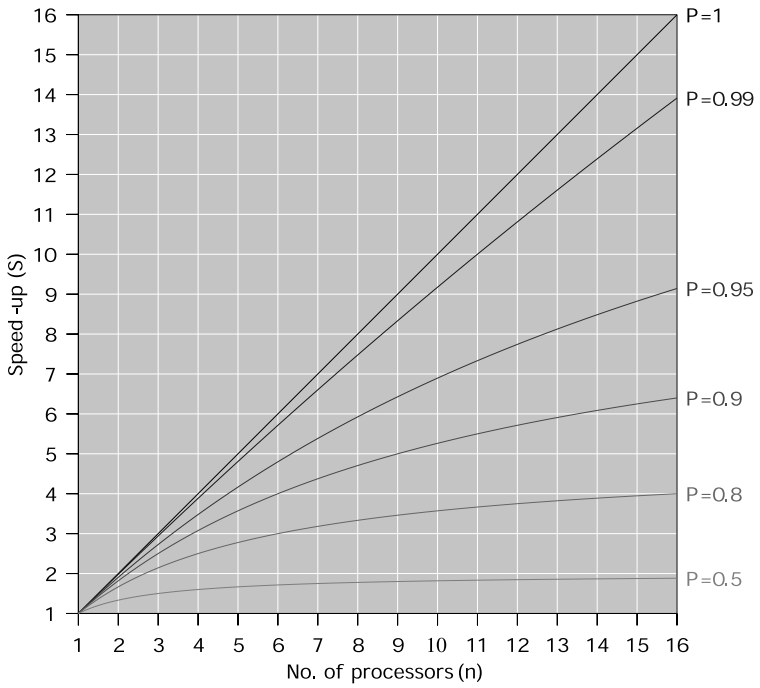


Figure 2.1: *Amdahl's Law: Parallel speedup (S) vs. parallel fraction (P).*

Implicit Parallelism

Writing programs without concern for how they map onto a given parallel architecture has obvious benefits. However, without explicity marking sections in the code as being parallel, a good automated strategy needs to be employed in order to decide when to fork parallel computation. Much work has been done in this area, with static parallelising compilers [122, 56, 16, 59] and also dynamic run-time recompilation techniques [136, 41, 172], ranging from conservative approaches [37, 121, 120] where all dependencies must be guaranteed before applying a given parallelisation technique to liberal speculative techniques [57, 135, 148] that speculate on dependencies, predict values and roll-back on dependency violations or value misspredictions.

Explicit Parallelism

Implicit parallelism, whilst being an attractive approach to locating parallelism, often under performs as it either extracts parallelism conservatively, missing potential performance gains, or it extracts parallelism too liberally and then wastes

time cleaning up speculative state changes. Additionally the performance of implicit techniques are often hard to reason without a solid understanding of the underlying compiler or runtime execution environment.

An alternative approach is for the programmer to annotate a program in order to indicate to the compiler and runtime execution environment that parallel computation might be beneficial. Most high-level programming languages support abstractions for implementing concurrency such as the concurrency package [94] in Java, and the POSIX thread library [21] in C and C++. Using such libraries it is possible for the programmer to abstract away much of the detail of the underlying implementation and concentrate on writing parallel applications.

Unfortunately explicit parallelism is often conservative because critical sections of code need to be locked, as incorrect locking can result in incorrect program execution, despite the fact that the majority of execution would succeed without the locks in place. Much work has been done looking at alleviating the cost of locks [133], lock-free synchronisation [86] and data structures [64] and more recently a resurgence in transactional memory [65, 61, 5, 134] where locking structures are removed entirely and the concept of a transaction is introduced. A transaction is a body of code that either completes, in which case all of its changes are committed, or fails and therefore none of the changes are committed.

2.1.3 Granularity of Parallelism

In general whilst parallelism is either implicit or explicity defined in an application, there also exists a range of granularity at which parallelism can be defined or extracted.

Instruction Level Parallelism

Arguably the finest grain of parallelism exploitable in an application is instruction level parallelism (ILP). ILP exists where multiple instructions within a sequence are independent and can therefore be executed concurrently. Figure 2.2 and 2.3 show two simple sequences of 3 instructions. In Figure 2.2 all three instructions are independent; there are no data dependencies between them, and they could all be executed in parallel. In Figure 2.3 however, the instructions must be executed

in sequence as the second instruction consumes the result of the first, and the third instruction consumes the result of the second.

```
LOAD R1, 32[R2]
ADD R3, R3, #1
SUB R4, R4, R3
```

Figure 2.2: *A simple code sequence amenable to ILP execution.*

```
ADD R3, R3, #1
ADD R4, R3, R2
STORE 0[R4], R0
```

Figure 2.3: *A simple code sequence containing no ILP.*

The amount of ILP within an application varies widely depending on the type of application. A large amount of research has been done looking at locating ILP in applications [77, 167, 166, 24], and exploiting ILP using both compile-time optimisations [99, 70] and architectural innovations [156, 100].

Data Level Parallelism

Data Level Parallelism (DLP), also referred to as SIMD [48], exploits parallelism in applications where a single operation is applied to multiple data sets concurrently. Scientific applications that work on massive vectors or matrices are often amenable to data-parallelism optimisations, as are many image and signal processing applications [151]. DLP is usually fairly fine-grained, as multiple data elements or registers are often processed using a single instruction and as such is usually exploited using instruction set extensions. Most modern microarchitectures contain vector specific instructions within their instruction set architecture, for example Intel's MMX [126] and later SSE(1–4), AMD's 3DNow! [118], PowerPC's AltiVec [39], Sun's VIS [157] and Hewlett-Packard's MAX-2 [96].

Figure 2.4, presents a loop from the MPEG encoder [110]. The loop is performing a sum of absolute differences calculation on the arrays `ref` and `curr`. After scalar expansion and loop fission the two loops shown in Figure 2.5 are formed. The first loop is then amenable to data level optimisations, as a subtraction is being applied on multiple datasets, in Figure 2.5 it is assumed that the underlying architecture has the capability to dispatch four substractions concurrently and so the loop has an unrolling factor of 4. Exploiting DLP on multimedia applications, such as MPEG4 encode/decode and H.264 decode has shown speedups in performance by upto 2 times [54, 95].

```
for(i=0; i<16; i++){
   localdiff = ref[i] - curr[i];
   diff += abs(localdiff);
}
```

Figure 2.4: *Inner loop of the motion detection algorithm used in MPEG encoding, calculation of the sum of absolute differences.*

```
for(i=0; i<16; i+=4){
   T[i+0] = ref[i+0] - curr[i+0];
   T[i+1] = ref[i+1] - curr[i+1];
   T[i+2] = ref[i+2] - curr[i+2];
   T[i+3] = ref[i+3] - curr[i+3];
}

for(i=0; i<16; i++){
   diff += abs(T[i]);
}
```

Figure 2.5: *Inner loop after scalar expansion and loop fission. The first loop is now amenable to data level parallelism optimisations.*

Loop Level Parallelism

Another form of parallelism exploited by both implicit and explicit optimisation techniques is Loop Level Parallelism (LLP) [3, 89, 12]. LLP fits into the MIMD category in Flynn's taxonomy [48]; the total iterations of a loop being divided amongst the multiple processors in a multiprocessor system, and executed concurrently. Until recently gains from LLP were limited because of fine-grain synchronisation and loop-carried dependencies [98], limiting the applicability to loops amenable to course-grained division. More recently however the reduction in communication latencies between multiple processors, especially those integrated in the same chip or multi-chip module, has allowed loop level optimisations to be applied on smaller loops exhibiting fine-grained parallelism [172].

Thread Level Parallelism

Perhaps the coarsest grain of parallelism exploitable within applications is Thread Level Parallelism (TLP). TLP is the parallelism inherent in an application that runs multiple threads of execution concurrently. TLP has traditionally been exploited in commercial applications, for example databases and web servers, where system input/output is a generally a limiting factor on performance. By running multiple threads in parallel, these applications are able to hide the latency incurred by the input/output, and therefore increase the overall throughput of

the application [85].

More recently with the advent of chip multiprocessors, exploiting TLP has become an important source of performance gain within the desktop market. Several studies have shown that upto 1.4 times speedup via TLP exists even amongst the threads within common desktop applications [46, 47], however the level of TLP present is only sufficient to provide performance speedups on dual or quad processor architectures.

Like ILP and LLP, TLP is also limited by data and control flow dependencies. In an attempt to overcome some of these dependencies and hence extract more TLP from existing applications much work has been done looking at speculating on data dependencies [148, 119, 146, 147, 173], a technique referred to as Thread Level Speculation (TLS). TLS allows sections of code that can not be guaranteed as independent at compile- or run-time to execute speculatively in parallel. State changes are locally buffered, and results are only committed when no other threads exist that can change the state previously seen by a speculative thread. Studies have shown the TLS can extract between 1.74–2.1 times speedup for floating point applications, and 1.23–1.7 times speed-up for integer applications [130, 74].

2.2 Architectural Parallelism

Having outlined in the previous section techniques for exploiting parallelism within applications, the following sections discuss how parallel execution is supported in hardware.

2.2.1 Bit Level Parallelism

Of all the hardware techniques that exploit parallelism, bit level parallelism is the finest-grained and is usually employed inside logic blocks within an architecture. Exploiting bit level parallelism, whilst not able to reduce the total number of cycles required to execute a given application, is often used to reduce the critical path and hence allows for increased operating frequencies.

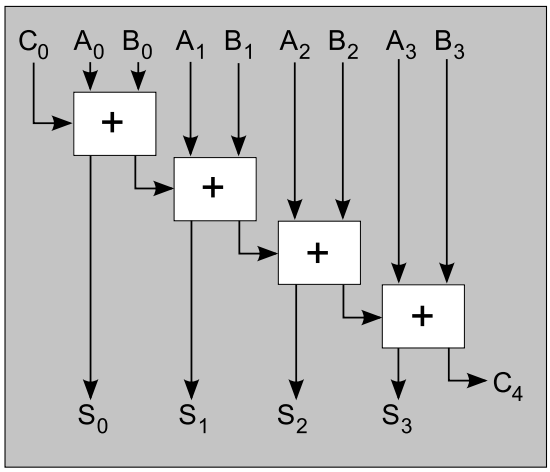


Figure 2.6: *4-bit ripple-carry adder.*

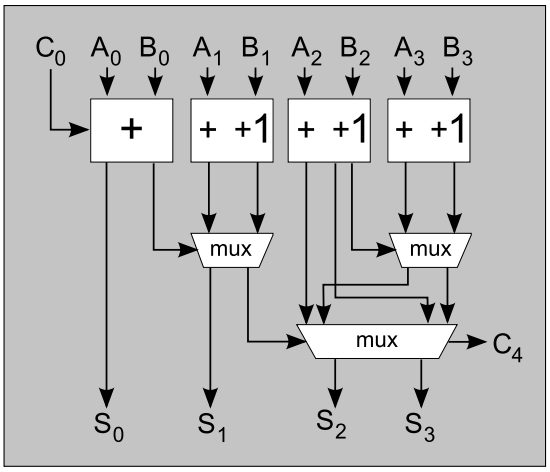


Figure 2.7: *4-bit carry-select adder.*

A simple example would be the implementation of a 4-bit full-adder which can be implemented as a ripple-carry adder, shown in Figure 2.6. In a ripple-carry adder all but the first in the sequence of the 1-bit adders must wait for a carry input before producing an output. In order to exploit bit level parallelism a carry-select adder can be used as an alternative, shown in Figure 2.7. In a carry-select adder two additions are computed for each bit pair, using a carry of 0 and 1. The correct result is later selected and propagated when the true carry value is known.

2.2.2 Data Level Parallelism

As mentioned previously, Section 2.1.3, additional SIMD instructions have been added to most modern microarchitectures to better support common multimedia algorithms. These algorithms generally consist of operations on values represented by 8-, 16-, and 32-bit integer or fixed-point data types, which are typically smaller than the maximum datapath width, 32- or 64-bits or greater. Also mentioned in Section 2.1.3, SIMD optimisations can account for 2 times speedup within multimedia applications, while the addition of SIMD support in hardware has been shown to require minimal additional logic, in the UltraSparc I this additional logic increased the die area by approximately 3% [158].

Figure 2.8 shows a high-level abstraction of the DLP within the data path used by the Streaming SIMD Extensions (SSE) instruction `psadbw`, within compliant

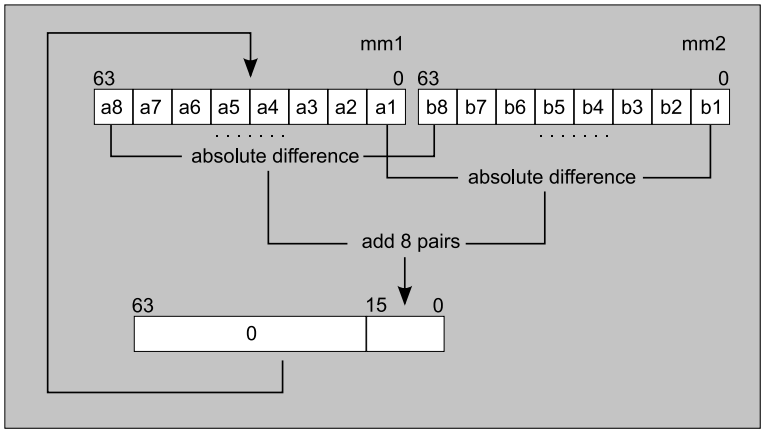


Figure 2.8: *Data Level Parallelism: the datapath inside the execution of the* `psadbw` *SSE2 instruction.*

Intel architectures. Eight bytes are packed into each of two 64-bit registers. The architecture is then capable of performing the sum of differences calculation on all eight pairs of bytes with a 4-cycle latency, replacing 8 subtractions and accumulations and handling the absolute value without using branch instructions. The loop presented in Figure 2.4, using the SSE2 instructions and associated hardware, can be reduced to just two `psadbw` instructions. Within the MPEG encoder's motion detection algorithm this can be used to produce a factor of 2 speedup [71].

One limitation of DLP support in hardware is that it is only invoked by an application that has already been compiled or coded to include the extended SIMD instructions, extracting DLP dynamically at runtime is impractical. This means that when extensions are added to an instruction set architecture, and extra logic blocks are added to the hardware, application code must be rewritten or recompiled in order to use it.

2.2.3 Instruction Level Parallelism

Section 2.1.3 introduced a simple sequence of instructions that is amenable to ILP. In order to illustrate how ILP can be exploited in hardware, it is first necessary to introduce a simple pipelined processor.

Simple Pipelined Processor

The simple pipelined processor, shown in Figure 2.9, is a simple MIPS-like [79], control-flow, load-store architecture with a 5-stage pipeline. The Fetch stage is responsible for loading instructions from memory and maintaining the current program counter. The Decode stage decodes the instruction, fetches register contents and determines when the instruction can be issued. The Execute stage performs the operation. The Memory stage is responsible for loading or storing data into the memory hierarchy. Finally the Writeback stage commits the results of the operation back into the registers.

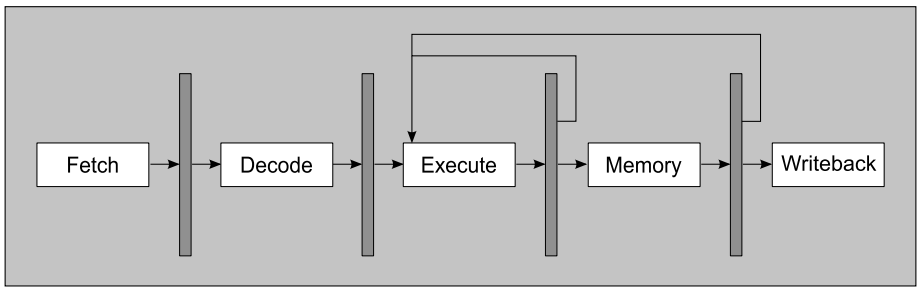


Figure 2.9: *A simple pipelined architecture: showing pipeline stages, latches and data-forwarding paths.*

In the simple pipelined processor, shown in Figure 2.9, ILP can not be exploited as there is no opportunity to execute multiple instructions concurrently, see Figure 2.11.

Superscalar and VLIW Processors

In order to exploit ILP a processor must be able to issue and execute multiple instructions per cycle from the same instruction stream. Two broad classes of architecture, VLIW [45] and superscalar [7, 124, 123] have been designed around the concept of multiple-issue pipelines, an example of which is shown in Figure 2.10.

Superscalar processors were originally developed as an alternative to vector processors, aimed at achieving vector processor performance but from exploiting ILP rather than DLP. In the pipeline of a superscalar processor upto n instructions can

be issued each cycle, as illustrated in Figure 2.11, equating to a maximum achievable throughput of n instructions per cycle. In an architecture where $n \geq 3$ and where the right functional units were available, all of the instructions in the code sequence listed in Figure 2.2 could be issued in a single cycle latency. Achieving 3 way ILP.

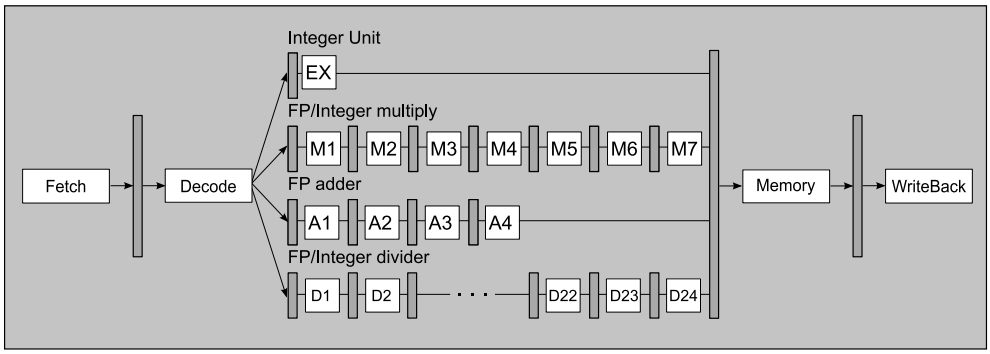


Figure 2.10: *A multiple-issue pipeline, with multiple functional units.*

In order to utilise an n issue superscalar processor efficiently one instruction needs to be issued to each functional unit in each cycle. Since the amount of ILP within a basic block[1] of code is small, instructions must be selected across basic blocks in order to keep the functional units busy. Compilers usually employ optimisations such as loop unrolling, code motion and register renaming, in order to exploit as much ILP as possible.

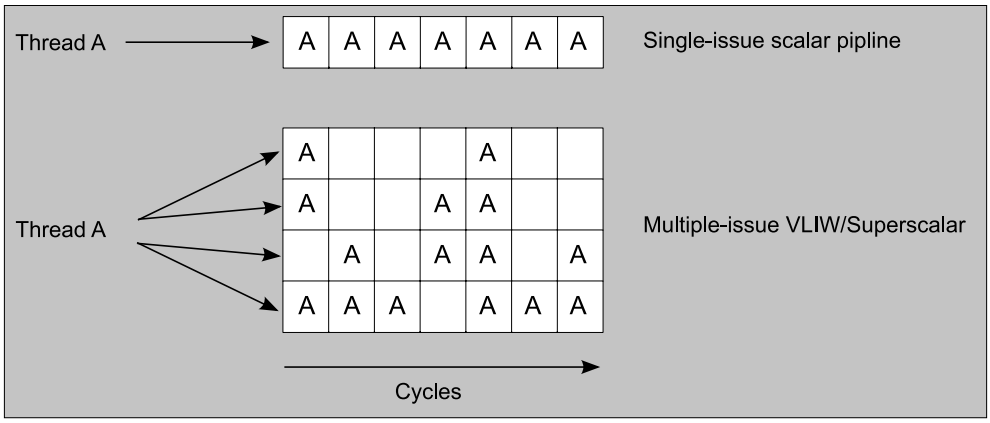


Figure 2.11: *Instruction scheduling within a single-issue and superscalar/VLIW processor.*

[1]A basic block is a sequence of code that has one entry point, one exit point, and contains no jumps or branches.

Superscalar processors can be subdivided by their scheduling policy. Statically scheduled superscalar processors are able to issue multiple instructions from a single instruction stream, where dependencies and hazards are resolved in the decode stage of the pipeline, such that the instructions are issued in the order they appear in the instruction stream. Dynamically scheduled, *out-of-order*, superscalar processors allow instructions to be issued in any order, as long as dependencies are maintained, regardless of whether preceding instructions have been issued. This simplifies the job of the compiler as some dependencies can only be determined at runtime, it allows the processor to tolerate dynamic cache behaviour and associated latencies and allows code compiled for a particular pipeline configuration to run efficiently on a different pipeline. The flexibility of dynamic scheduling results in further exploitation of ILP and therefore higher performance, than statically scheduled superscalar or VLIW processors, but requires substantial increases in the complexity of the hardware.

This complexity of dependence resolution amongst the inputs into and outputs from the multiple functional units that form a superscalar architecture increases quadratically with the number of functional units. This combined with diminishing returns from attempting to exploit wide-issue ILP have limited most superscalar architectures to between 2- and 8-way issuing.

2.2.4 Multithreading

In single-issue, VLIW and superscalar architectures multithreading, that is switching between multiple instruction streams, is achieved through software support of either cooperative or preemptive scheduling. These scheduling schemes rely on either the operating system or the application itself to concede use of the processor to another thread at regular intervals. In order for this to happen the running thread is required to save all of its current working state[2] into memory before the context can be switched. The replacement context is then required to load all of its previous working state into registers prior to continuing its execution of a thread. These software based context switches take many hundreds of cycles to complete and are only invoked once a thread has had many thousands

[2]The working set includes the registers and context specific state, such as the program count, stack pointer.

of cycles of processor use. The combination of the memory gap, as mentioned in Section 1.2.2, and increasingly multithreaded workloads, created the demand for processors capable of switching between one thread of execution rapidly to hide the memory latency of another. These multithreaded architectures are capable of issuing instructions from multiple threads, and can therefore exploit TLP during otherwise wasted memory delay cycles. The manner in which threads are scheduled for execution within a multithreaded processor defines two types of multithreading, *Interleaved MultiThreading* (IMT) and *Block MultiThreading* (BMT).

Interleaved Multithreading

IMT, also referred to as fine-grained multithreading, schedules an instruction from each of the architecturally supported thread contexts each cycle, see Figure 2.12, leading to a context switch delay of zero cycles. Initially the number of threads supported by IMT processors was equal to the number of pipeline stages. This eliminates control and data dependencies between pipeline stages and removes the need for complex hardware interlocking or data forwarding. Without these complex paths the critical paths in the pipeline are reduced and it can therefore be clocked at a higher frequency. A disadvantage to the original interleaved multithreading scheme as implemented in the HEP architecture [142], is that single-threaded execution was poor as a thread could only utilise the pipeline every n cycles, where n is the number of hardware supported threads. Two techniques proposed in the literature overcome this limitation, the first *dependence look-ahead* [155] allows the compiler to tag each instruction with a number of bits informing the pipeline how many following instructions are independent allowing each context to gain more continuous time in the pipeline. The second technique *Interleaving* [92], added data forwarding paths and hardware interlocking allowing contexts to be switched on a cycle by cycle basis, but also efficiently supporting single-threaded execution. This form of multithreading is implemented in Sun's Niagara architecture [81].

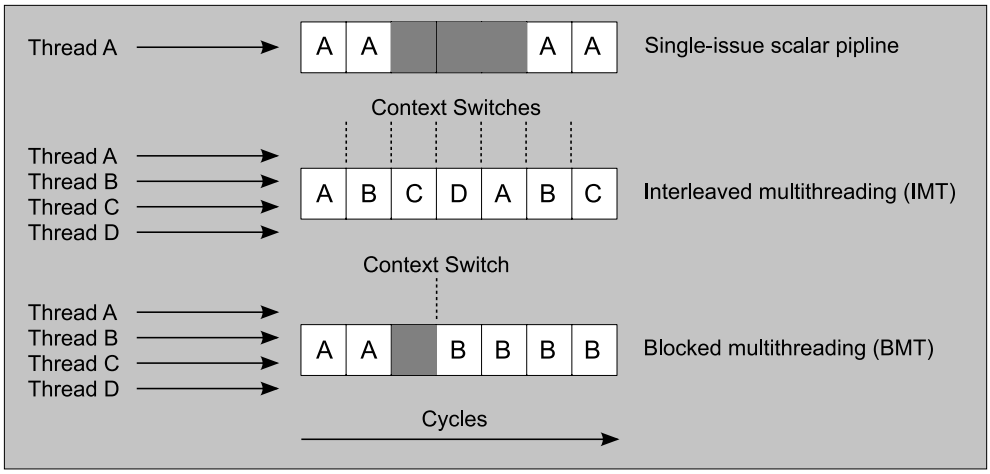


Figure 2.12: *Scalar multithreading scheduling schemes.*

Block Multithreading

BMT, also referred to as coarse-grained multithreading, schedules instructions from a fixed context until a specific event triggers a context switch, usually a long-latency operation. When only 1 context is runnable no context switches are triggered and the context achieves good single-threaded performance. The trigger event can either be *static* or *dynamic*. A statically triggered BMT processor switches contexts when a particular instruction is issued. These instructions can either be specific context switch instructions or instructions within the instruction set that are likely to cause long latency stalls, such as loads, stores and branches. Dynamically triggered BMT processors switch context when a dynamic event occurs such as a cache miss, see Figure 2.12, an interrupt or after the context has run for a given quantum, say a thousand cycles. The advantage of static BMT is that the cost of the context switch is still minimal, zero or one cycle, whereas dynamic multithreading requires the pipeline to be flushed causing multiple delay cycles.

In general BMT is less efficient than IMT especially when used in deep pipelines [91], due to the cost of flushing pipeline stages following a context switch.

2.2.5 Simultaneous Multithreading

Simultaneous MultiThreading (SMT) [159] is the natural hybrid formed by including multithreading techniques within a superscalar architecture. A SMT processor is capable of issuing multiple instructions from multiple threads each cycle, as shown in Figure 2.13. Each hardware context can compete each cycle for all of the available functional units, allowing both ILP and TLP to be exploited, increasing pipeline utilisation and hence overall performance.

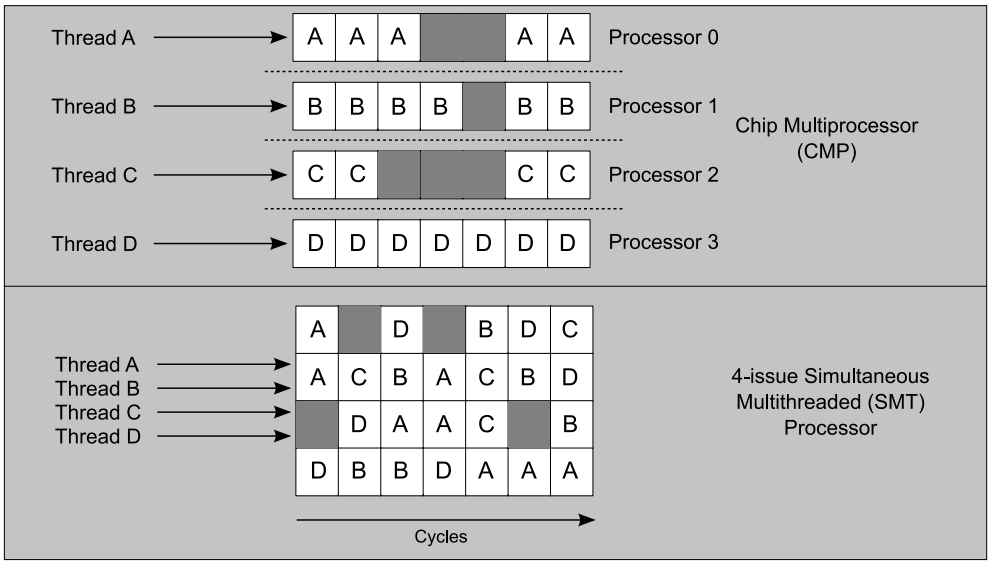


Figure 2.13: *Instruction scheduling within an SMT processor and a CMP.*

2.2.6 Chip Multiprocessors

As the number of transistors integrated onto a single silicon chip has increased, the ability to integrate multiple complete processors and a memory hierarchy on the same chip has become feasible. Many Chip MultiProcessor (CMP) designs have been proposed [44, 60, 82, 32, 103, 143, 145, 148, 13] and more recently implemented [81, 127, 78].

CMPs initially consisted of multiple single-issue processors, which were able to exploit TLP, but could not hide the latency of memory accesses amongst each thread. As the technology has allowed, however, the processing cores used have become more complex, and now multithreaded and even superscalar cores are

replicated on a single chip, allowing the architecture as a whole to exploit TLP, ILP and even memory level parallelism (MLP).

CMPs have two distinct advantages over a SMT, VLIW and superscalar architectures capable of exploiting the same level of parallelism. Firstly the design is less complex, each processor integrated on the chip can be fairly simple and is replicated. If necessary much of the complication of superscalar and SMT architectures can be avoided by opting for multithreaded scalar processors. This simplicity allows for higher clock frequencies and eases design validation. Secondly the power requirements within a chip multiprocessor are often far lower than the equivalent performing superscalar processor [108].

These advantages are balanced against the necessity to maintain memory coherency across multiple processors, requiring cache coherence protocols and shared or distributed memory hierarchies, which increase the complexity of cache control logic, however not to the same extent as the additional logic requirements of a centralised superscalar wide-issue design.

2.3 Summary

This chapter has reviewed the availability of parallelism within software and the exploitation of parallelism within processor and architecture design. ILP has been shown to provide benefits in the execution of single-threaded applications but is limited by the availability of independent instructions within those applications and the poor scalability of the complex control structures required to exploit it within processors. TLP, on the other hand, can be explicitly defined and extracted at multiple levels of granularity, and as such is easier to exploit and is more abundant within modern workloads. Architectures that exploit TLP can do so by hiding memory latency or by truly executing multiple threads in parallel using an SMT or CMP architecture.

The next chapter reviews the Jamaica CMP architecture, introduces the `jamsim` simulation platform developed to simulate it, and briefly describes the software environment used to run applications on the Jamaica architecture.

CHAPTER 3

Jamaica Chip Multiprocessor and Software Environment

In the previous chapter several techniques for extracting parallelism from application code, and methods for exploiting this parallelism within processor architectures were discussed. This chapter reviews the Jamaica CMP architecture, a CMP with multithreaded cores and hardware support for lightweight work distribution. The architecture is able to exploit thread level parallelism and hide the latency of memory operations increasing parallel throughput.

The Jamaica CMP architecture provides a base design for the work presented in subsequent chapters.

3.1 The Jamaica Chip Multiprocessor

The Jamaica[1] architecture [170], as shown in Figure 3.1, is a CMP architecture consisting of N cores integrated on a single chip implemented in a cycle accurate simulation platform. Each core has its own Level 1 (L1) instruction and data cache. These private caches are linked by an on-chip bus, employing the

[1]Jamaica is an acronym for **JA**va **M**achine **A**nd **I**ntegrated **C**ircuit **A**rchitecture.

MOESI coherence protocol, to a unified Level 2 (L2) cache and on-chip memory controllers. Bus snooping is used to implement load-linked and store-conditional instructions, as per the Alpha architecture [140], which support the implementation of critical sections within application code. Several additional features of the architecture aid the execution of object-oriented and multithreaded codes.

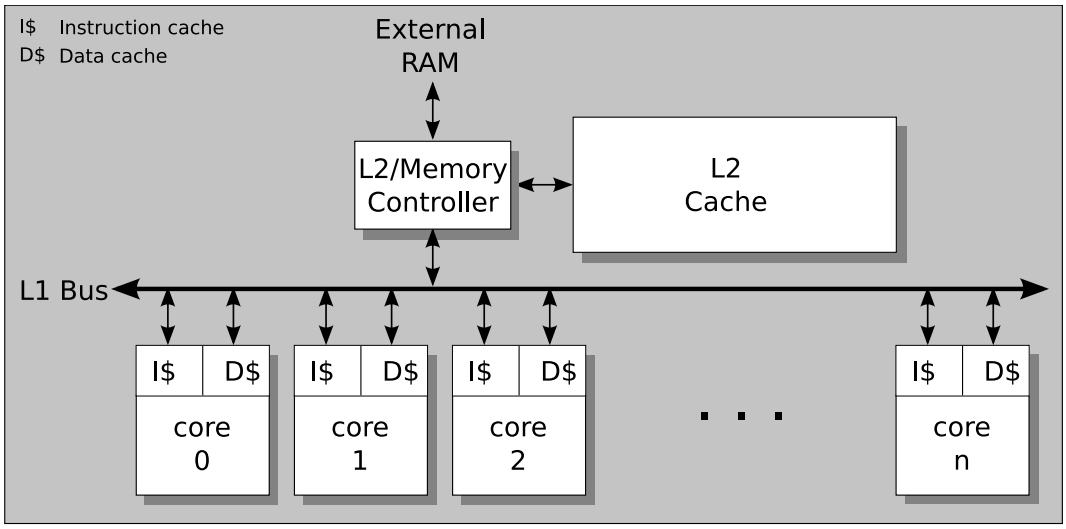


Figure 3.1: *The Jamaica single chip multiprocessor.*

3.1.1 Multithreading

Unlike many other CMP architectures [60, 13] each core within Jamaica is multithreaded to improve overall throughput. Multithreading gives the appearance of having multiple virtual processors per core by supporting multiple thread contexts in hardware. Each context maintains a set of context specific registers, containing register, interrupt and other thread specific state. Each context shares the core pipeline and L1 private instruction and data caches, illustrated in Figure 3.2.

The contexts within a core can reside in one of five possible states: *runnable*, *stalled*, *waiting*, *empty* and *idle*. The transitions between the five states is shown in Figure 3.3. The fetch stage of the core pipeline is responsible for fetching an instruction for the currently *active* context, chosen from the list of *runnable* contexts.

Jamaica employs a blocked, switch-on-cache-miss multithreading policy [161], extended by an additional switch-on-timer policy. The switch-on-timer policy

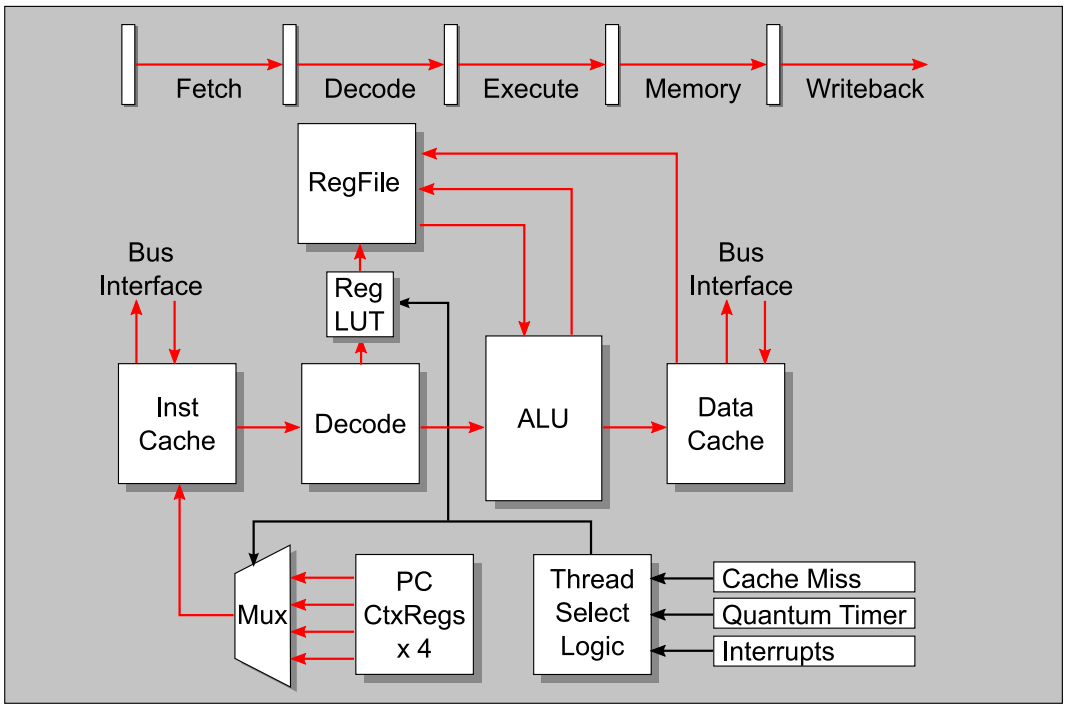


Figure 3.2: *Jamaica core: Multithreaded pipeline and support structures.*

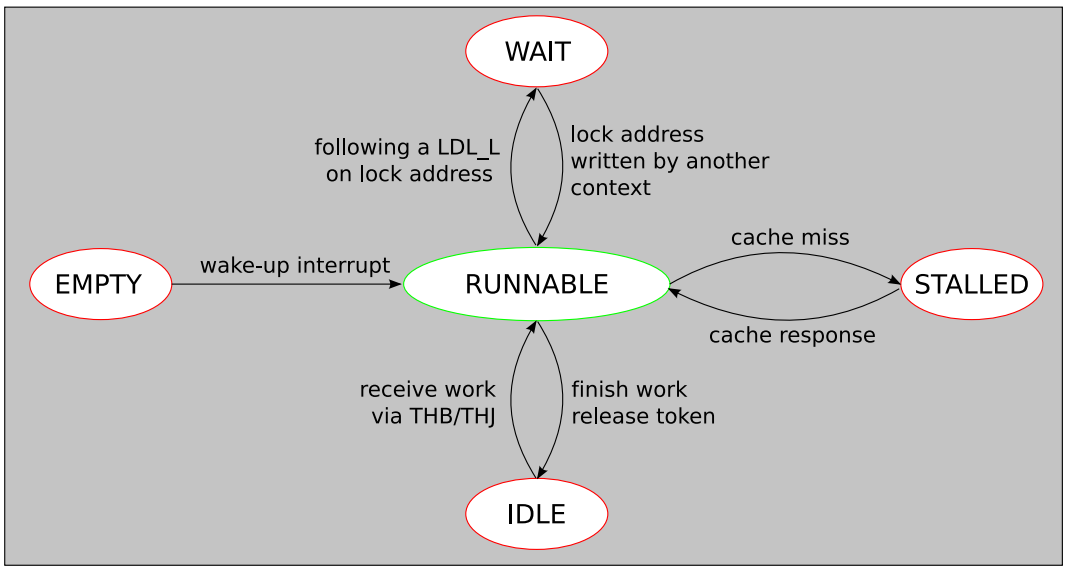


Figure 3.3: *Jamaica core: Context running states.*

triggers a context switch event when a context has been running for 1,024 cycles unhindered by cache misses. This maintains forward progress in the presence of spin-locks and the absence of an implicit context switch instruction. A round robin policy is used to rotate the *active* context from the list of *runnable* contexts for scheduling into the pipeline. If no contexts are *runnable* at a context switch event, the core itself becomes idle. A *stalled* context becomes a candidate for scheduling once the stalled memory access is resolved.

Context switching can help to hide memory latency, by keeping the core busy executing instructions from a runnable context during the memory stall incurred by another context, improving the overall throughput. The policy employed by Jamaica is most efficient when contexts suffer regular but not frequent cache misses. In the presence of only one *runnable* thread no context switching occurs.

3.1.2 Register Windows

To reduce the effects of frequent method calls within modern object-oriented languages [160, 36], a large windowed register file is shared between all of the contexts in a Jamaica core. The hardware supporting the register file implements a register windowing scheme [125, 131, 52], based on the multi-windows proposal [138]. The compiler can see 32 registers which are divided into four *windows* each containing eight 32-bit registers.

- (`%g0 - %g7`) *Global* window, shared by all contexts on a core.
- (`%x0 - %x7`) *Extra* window, private per context, statically allocated, non-volatile across methods calls.
- (`%i0 - %i7`) *In* window, private per context, dynamically allocated at each method call, volatile across method calls.
- (`%o0 - %o7`) *Out* window, private per context, dynamically allocated at each method call, volatile across method calls.

All the contexts on a core share the *Global* window, which is mapped directly to the bottom eight physical registers[2]. Each context has a private *Extra* window, mapped into the physical register windows located directly above the *Global* window. The *In* and *Out* window are allocated and released dynamically during method calls. When a context is in the *idle* state it consumes only two windows in the physical register file for the statically allocated *Extra* window and the bottom allocated *In* window, allowing all other windows to be allocated to runnable contexts.

Calling Convention

Although compiler techniques, based on register colouring [31, 132], can reduce the overheads associated with method calls, supporting register windows reduces the need to save and restore registers on each call and return, and is still useful for simplifying calling conventions. In Jamaica the *Out* registers of the caller method overlap with the *In* registers of the callee method and so passing small numbers of variables, six or fewer[3], is handled implicitly, as illustrated in Figure 3.4. Passing more values requires spilling and filling to the stack as per architectures not supporting register windows.

Register Mapping

One of the disadvantages of register windows is that window indices, in Jamaica a 5-bit register index encodes the window indices, must be mapped to a register in the physical set within the critical path of the decode stage which can prevent the pipeline from being clocked at higher frequencies [125]. In Jamaica the register operands decoded from an instruction are translated to physical addresses by a context Look-Up-Table (LUT), illustrated in Figure 3.5.

Downstream of the decode stage, all register indexes are physical and therefore data hazards and forwarding can occur using the same forwarding path and detection logic found in architectures containing flat register files.

[2]Additionally %g0 is hardwired to the value 0.

[3]Only 6 registers are available for passing values as two registers are explicitly used to pass the return PC, and the stack pointer.

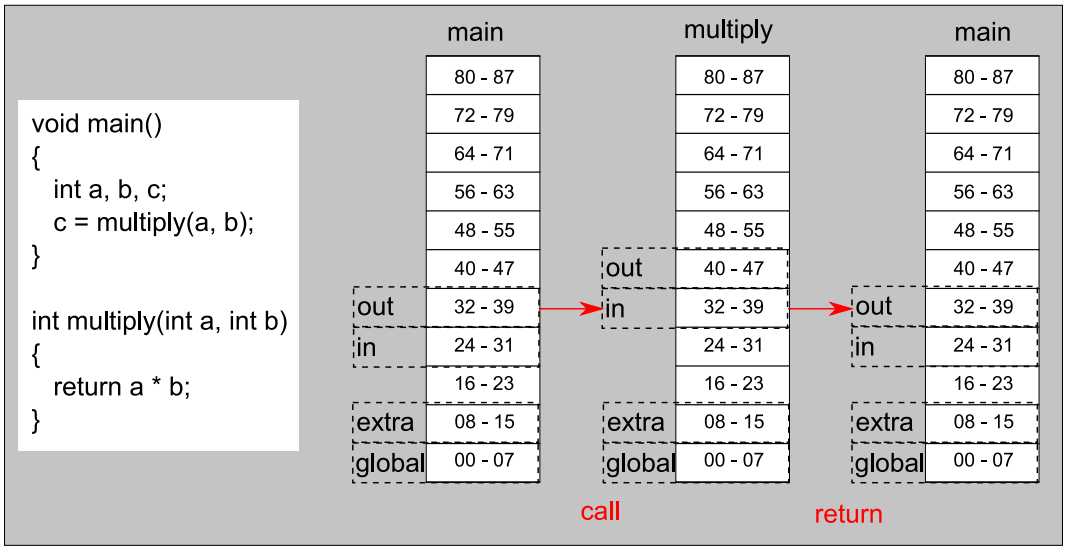


Figure 3.4: *Jamaica core: Register windows, call and return overlaps.*

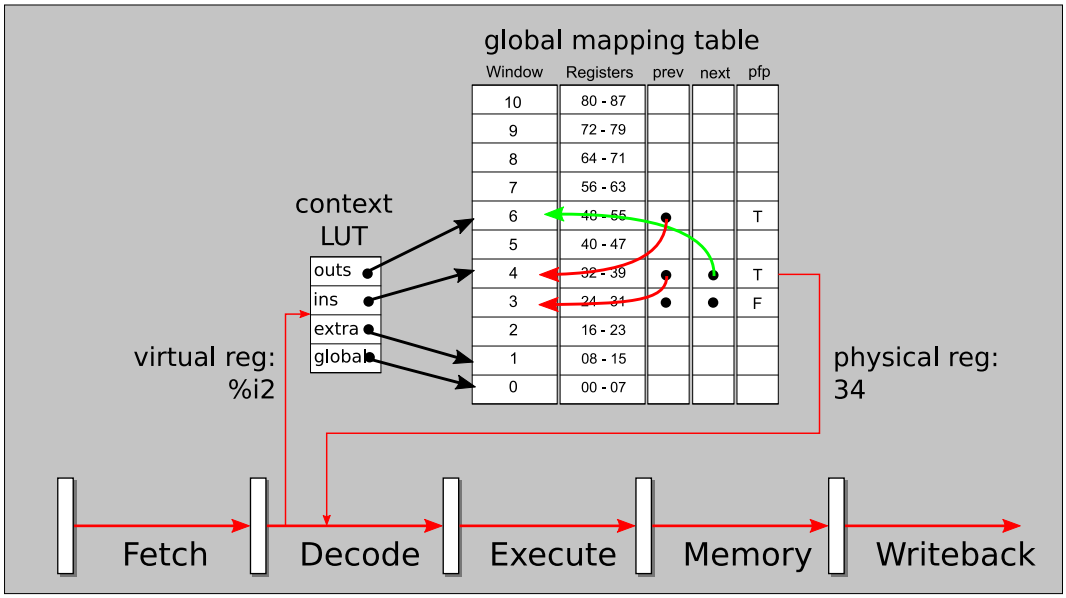


Figure 3.5: *Jamaica core: Register windows; virtual to physical register lookup.*

Window Management

Dynamic allocation of a new *Out* window occurs during a method call by setting a pointer from the previous *Out* window to a currently free window. The free window is then allocated and can not be used until it is released by a return. Window allocation is not restricted to contiguous windows and so a backward, *prev*, pointer is stored to trace back to the previous window on a return. A forward, *next*, pointer is also stored. This is necessary as a window can be evicted to the stack when all windows are allocated and a context attempts a call. When such an eviction occurs a previous frame present flag (pfp) is cleared to indicate that the window has been spilled into memory.

These pointers are stored alongside the index mappings in a global mapping table, shown in Figure 3.5.

3.1.3 Lightweight Task Distribution

A novel feature of the Jamaica architecture is the hardware support for lightweight task distribution. This hardware support consists of a ring interconnect connecting all of the processing cores. The ring allows *active* contexts to locate *idle* contexts to which tasks can then be distributed.

Idling Contexts

When an *active* context exits from the bottom of its current executing stack, detected in hardware by a return from an *In* window that has no predecessor, the context's state changes from *runnable* to *idle* and a token is inserted into the sequence of tokens circulating around the ring interconnect. There is no software teardown prior to the release of the token, which leaves the resident software stack in place in the context specific registers, ready to run when work is distributed to the idle context. The token placed onto the ring simply contains the identity, a unique `contextId` stored in a hardware fused register, of the context now in the *idle* state.

Two additional instructions within the Jamaica instruction set are used by an *active* context to locate an idle context, by requesting a token from the interconnect

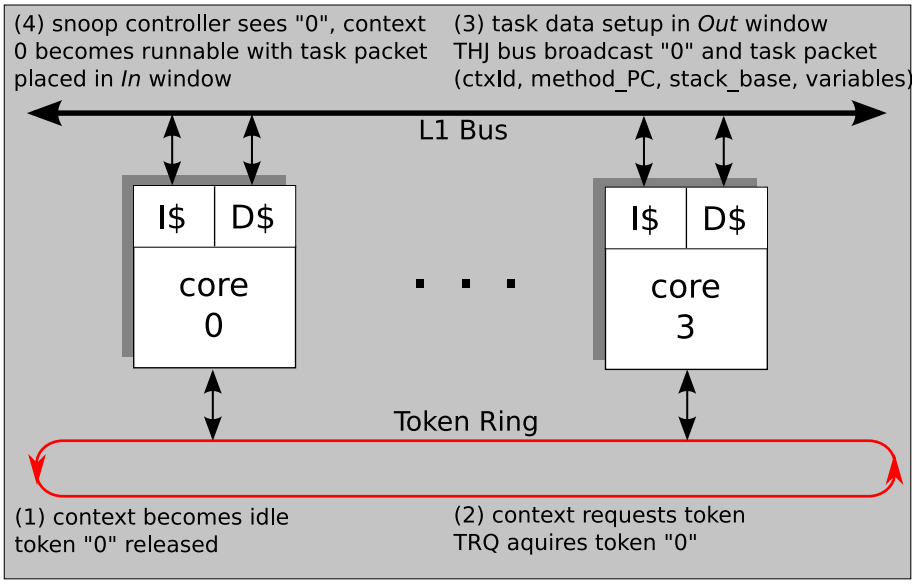


Figure 3.6: *Jamaica core: Lightweight task distribution.*

using a token request (`TRQ`), and then to distribute work to that context using a thread jump (`THJ`). The simple calling convention, mentioned in Section 3.1.2, is also used during task distribution, the *Out* window being used to hold task setup information as well as task input variables. When the `THJ` instruction executes, a transaction is placed onto the shared bus, and the relevant snoop controller wakes up the idle context, see Figure 3.6. If when executing the `TRQ` no other contexts are idle, or a token is not acquired from the ring within a given number of cycles, the `TRQ` instruction fails and the *active* context processes the work locally.

The average latency to find an idle context, if one is present, on an N core Jamaica architecture is $\frac{N}{2}$ cycles, which for small numbers of N is significantly lower than locating idle processors through a shared locked queue stored in memory. Additionally the ring interconnect can be accessed in parallel, allowing multiple cores to either release or request tokens concurrently.

3.1.4 Branch Prediction

Branch prediction in the Jamaica core is implemented using a simple 2-bit saturating up/down counter policy [171], indexed from a Least Recently Used (LRU) evicted branch history table. The table is accessed during the fetch stage as part of the calculation of the next PC. A hit in the branch history table alters the PC

according to the 2-bit status. The branch is subsequently evaluated in the execute stage of the pipeline, where the table is updated accordingly. Miss-predictions are handled by flushing the pipeline and setting the PC to the correct branch target.

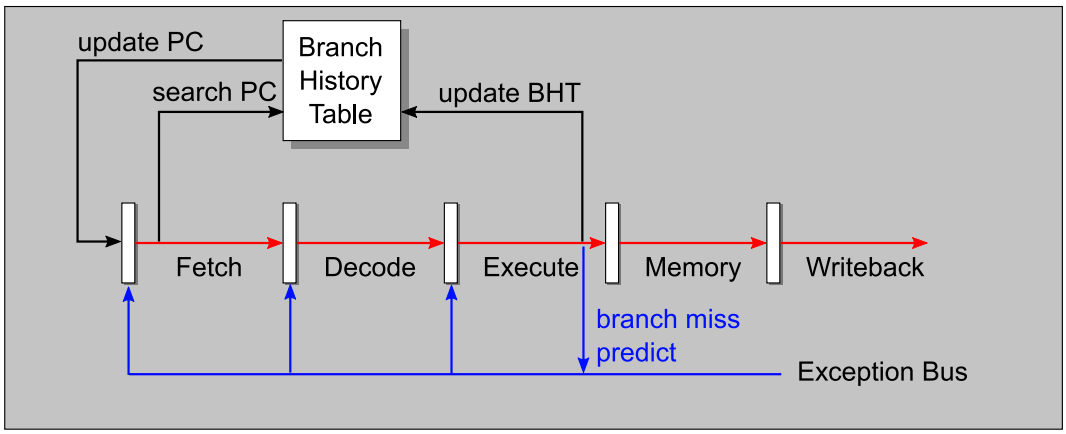


Figure 3.7: *Jamaica core: branch prediction.*

In the Jamaica core, the delay between speculating on the branch target and subsequently calculating the true target is only 2 cycles and so a context switch is not triggered during this period. Branch prediction is however essential for keeping the pipeline busy between context switches.

3.1.5 Coherent Shared Memory Hierarchy

The Jamaica CMP has private L1 instruction (I$) and data caches (D$), connected via a shared bus to a single shared L2 cache. Access to external memory is through the integrated L2 cache and memory controller. All caches in the architecture maintain sequential consistency to allow for standard shared memory programming.

Cache Coherence

All caches in the Jamaica CMP are kept coherent by snooping a shared L1–L2 bus and cooperatively implementing a version of the Modified, Owned, Exclusive, Shared and Invalid (MOESI) cache coherence protocol [153]. A cache line in the Modified or Exclusive state is writeable, since it holds the only valid copy of

data present in the system, the Modified and Owned states indicate that the cache holding the line is responsible for writing the only updated dirty data back to memory. The protocol used in Jamaica is based on features taken from the Firefly, Dragon [8] and CRAC [154] protocols. In particular L1 cache to cache transactions are allowed without updating memory, and ownership can be transferred without informing the L2 cache or memory.

L1–L2 Shared Bus

All private L1 caches and the shared L2 cache and memory controllers are connected by a shared bus. The bus implements a protocol allowing split transactions and transaction pipelining similar to the SGI Challenge's Power2 bus [51]. Each transaction is split into two distinct phases, request and response, allowing the memory hierarchy to service requests during the delay due to a request missing in the L2 cache and being serviced from main memory.

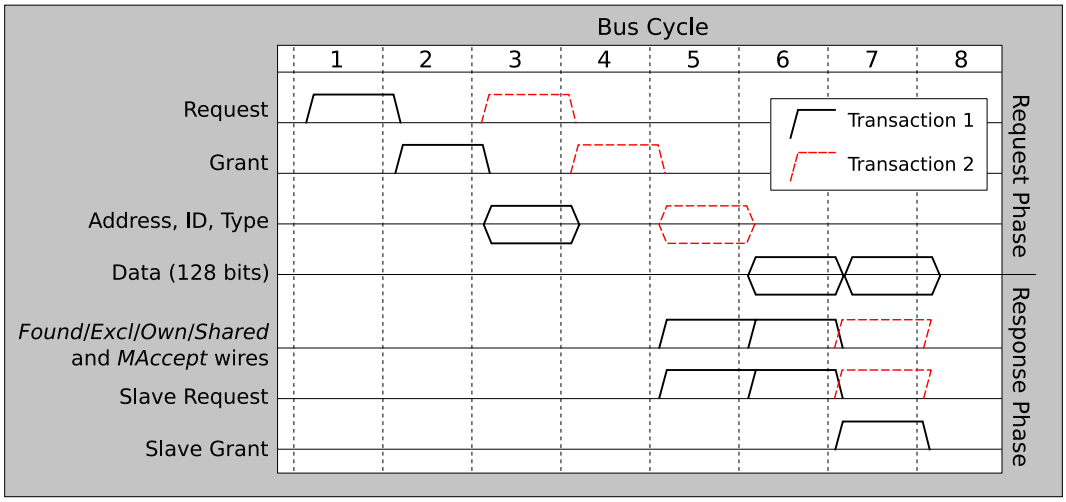


Figure 3.8: *Jamaica: Split transaction bus protocol.*

The Jamaica shared bus implements an 8-cycle protocol, illustrated in Figure 3.8. The protocol allows pipelining of multiple requests, each starting at 1 cycle intervals, with the condition that two requests for the same address can not occur in sequence. This condition is required as the data and tags are in an unstable state during cycles 6 and 7, the same cycles in which a subsequent transaction would be checking the state of the corresponding cache line.

Priority on the shared bus is given to requests originating from the L2 cache, all other requests for access to the bus are fairly arbitrated by selecting the least recently used cache. Requests that can not be responded to within the 8-cycle protocol, those that miss in all L1 caches and the L2 cache, are responded to by the L2 cache after the data is fetched from memory. The request and response pairs are matched using the *ID* placed on the bus in cycle 3, a sequential number unique to each cache.

Level 1 Private Caches

Each L1 private cache is shared by the multiple contexts supported by each core. When a memory access misses in an L1 cache a context switch is triggered, as mentioned in Section 3.1.1, and an entry is placed in the cache request table, shown in Figure 3.9. The request table is responsible for progressing outstanding requests onto the shared bus and for handling the responses. A small number of buffers are provided for lines requiring writeback in the Owned or Modified state. The writeback buffer is also used as a small victim cache when unmodified data is evicted.

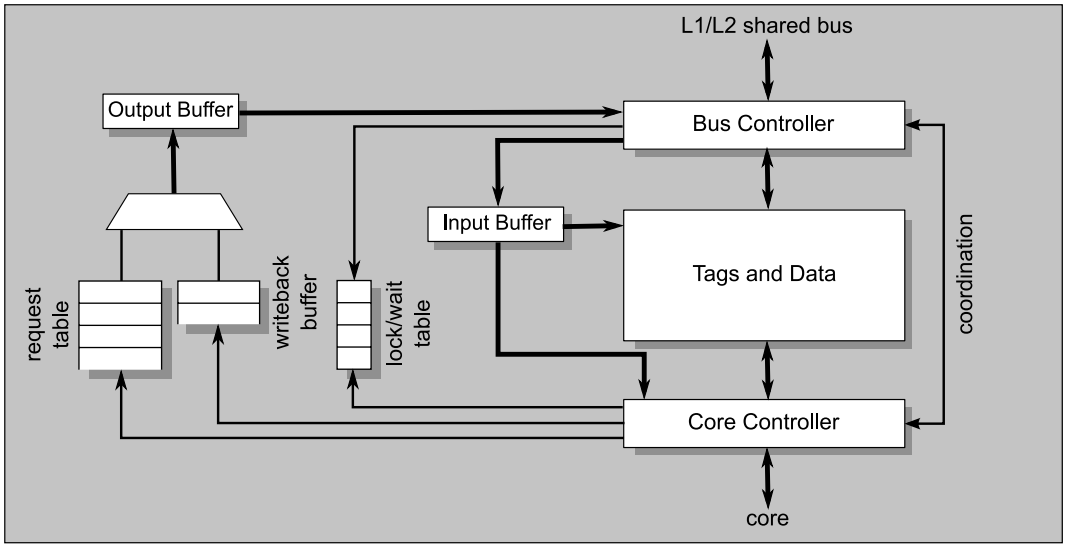


Figure 3.9: *Jamaica: Level 1 private cache.*

Level 1 Atomic Primitives

Atomic primitives, used to enforce critical code sections, are implemented using load-linked (LDL_L) and store-conditional (STL_C) pairs, as illustrated by the code sequence Figure 3.10. Inside each L1 cache a small lock table is maintained, containing one address and lock flag per context. The flag is set as part of the LDL_L operation. A subsequent STL_C is allowed to complete, committing the data word and setting an acknowledgement value of 1 in an allocated register, if the lock flag is still set. A write to the lock address by any other context in the architecture in the intervening period resets the lock flag, and the STL_C fails, writing a value 0 into the allocated register.

```
lock_aquire:
        LDL_L %i1, 0(%i0)        ! load lock
        BNE %i1, wait_release     ! wait if non-zero
        ADD %g0, 1, %i1
        STL_C %i1, 0(%i0)        ! try to acquire
        BEQ  %i1, lock_aquire ! try again if store failed
        RET
wait_release:
        WAIT !wait for lock release
        BR lock_aquire !retry lock aquire
```

Figure 3.10: *Jamaica: Lock acquisition code.*

Level 1 Synchronisation

For synchronisation the LDL_L can also be paired, in the Jamaica instruction set, with the WAIT operation, also shown in Figure 3.10. After setting the lock flag with a LDL_L the WAIT instruction sets the context state to *wait*. The context is unavailable for scheduling until the lock address is written to by another context. In practice the WAIT operation is used sparingly, however, as multithreaded applications containing more software threads than hardware supported contexts require that multiple threads are serviced by each context, and disabling a context with a WAIT could lead to starvation and dead-lock.

Level 2 Shared Cache

In the Jamaica CMP the L2 shared cache is less complex than the L1 caches. The L2 cache is unified, and is accessed via the shared L1–L2 bus. The L2 cache only responds to bus transactions for which the data is not contained in the L1 caches. This occurs when a transaction is in progress and during cycle 5 of the bus protocol, see Figure 3.8, the *Found* wire is not set high. The L2 cache either responds with the data in following cycles by setting the *Found* flag and state flags (*Excl*, *Owned*, *Shared*) high, or sets the memory accept (*MAccept*) flag to high in which case the request is forwarded to memory by the L2 controller, see Figure 3.11.

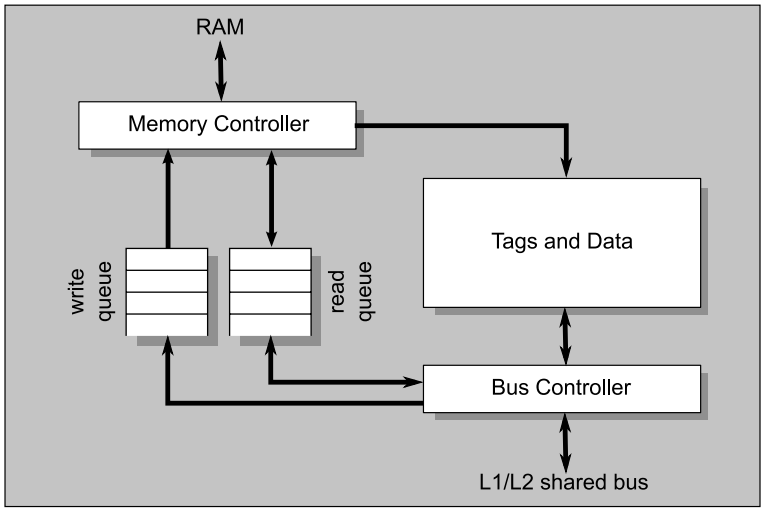


Figure 3.11: *Jamaica: Shared level 2 cache and memory interface.*

If a bus request for a line not in the L1 or L2 caches occurs when the memory queues are full, then neither the *Found* nor the *MAccept* wire are set high. The L1 cache must subsequently re-attempt the same request after a subsequent successful arbitration for the bus.

3.1.6 Hard and Soft Interrupts

Support is provided within Jamaica for handling a limited number of hardware and software generated interrupts. Interrupts vector a contexts execution path to handler code located at the bottom of memory, addressed by the type of

interruption. A software interrupt, `SIRQ`, can be delivered to any context in the *runnable*, *waiting*, *stalled* or *empty* states. The `SIRQ` is delivered to a context by the shared bus in a similar manner to the `THJ`, without a data payload. Contexts in the *idle* state can only be restarted using a `THJ`, therefore a `SIRQ` to an *idle* context is ignored.

Jamaica currently employs a single software interrupt, used to wakeup all contexts and vector them to a boot code sequence in order to setup a minimal stack, and a single hardware interrupt to trap on accesses to invalid memory addresses.

3.1.7 Devices

As Jamaica is currently only a simulated architecture, there is no defined device interface, and hence no associated device hardware interrupts. The simulation of Jamaica enables calls to the underlying operating system, for I/O operations through a set of defined built in operations. These operations are called within the architecture by subroutine jumps, `JSR`, into small negative addresses. The simulator recognises this address range and the calls are bypassed through to the underlying operating system upon which the simulation platform is running.

3.2 Jamaica Core Revisions

Having outlined the Jamaica CMP architecture in Section 3.1, this section describes several revisions that this thesis has made to the core to improve and simplify the architecture.

3.2.1 Interleaved Multithreading

As mentioned in Section 3.1.1 the Jamaica core architecture supports the execution of multiple hardware-supported contexts within the same pipeline using blocked switch-on-cache-miss multithreading. Interleaved execution of multiple contexts within the Jamaica architecture has been added providing further support for the execution of fine-grained threads.

The IMT policy is similar to the scheme presented by Laudon *et al.* [92] and the scheme implemented in the Niagara (Sparc T1) architecture [81, 152]. The *active* context is selected for execution every cycle from the set of *runnable* contexts. This allows multiple contexts to inhabit the pipeline concurrently and requires additional exception handling logic to determine the context from which exceptions are triggered.

Additionally the policy adds another context state, *long_latency_stall*, similar to the `LLI` state in the T1 architecture [152]. This state inhibits the context from being scheduled during operations that require multiple cycles. These operations include `TRQ`, `JSR`, `BSR` and `RET`. In the execution of the `TRQ` instruction, a configurable number of poll-cycles is included as an instruction operand, during which the token interface unit is able to poll the token ring for free context tokens. Rather than stalling the whole pipeline during these poll-cycles, or executing multiple `TRQ` instructions, a context executing a `TRQ` instruction is set to *long_latency_stall* and only re-scheduled after either a token is located or the poll-cycles expire. During this polling period, the pipeline is available to other runnable contexts. The following section describes the operation of the `JSR`, `BSR` and `RET` operations.

3.2.2 Working Set and Register Windows

As outlined in Section 3.1.2, the windowing scheme employed by the Jamaica architecture adds considerable complexity into the critical path of the decode stage in the pipeline. To reduce this complexity and maintain compatibility with the Jamaica instruction set and associated software, the register windowing scheme has been greatly simplified by removing window management and offset indexing from the decode stage.

During normal execution a context accesses registers `%g0-%x7`, from a set of 32 working registers indexed directly. When a call (`JSR` or `BSR`) or return (`RET`) is decoded in the decode stage the context is placed into the *long_latency_stall* state. During the subsequent stall period, window management occurs and other runnable contexts can be scheduled into the pipeline. It is anticipated that window management can occur within two-cycles, using a 16-wide register transfer port, labelled (1) in Figure 3.12 and (3) in Figure 3.13, and so in the revised

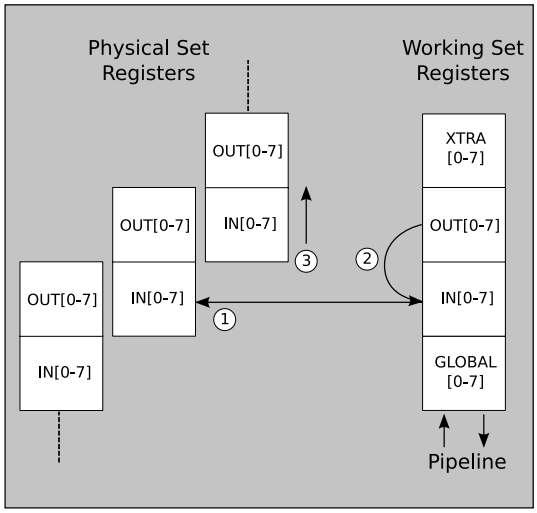


Figure 3.12: *Window Call: (1) write IN to physical, (2) copy OUT into IN, (3) increment window pointer.*

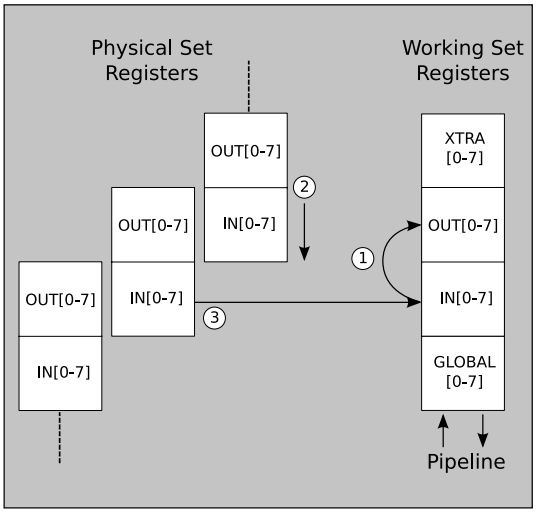


Figure 3.13: *Window Return: (1) copy IN into OUT, (2) decrement window pointer, (3) read IN from physical.*

Jamaica core a `JSR`, `BSR` or `RET` incurs a stall latency of 2 cycles before becoming available for rescheduling.

Implementing register windows in this manner has been shown to reduce the overall footprint as all but the working set of registers can be implemented in compact 6-transistor per bit SRAM cells and decreases the critical access time to the working set of registers [81].

3.3 Jamaica Software Environment

As the Jamaica instruction set is significantly different from both the Alpha instruction set, from which many of the instruction formats evolved, and from other common instruction sets, the Jamaica architecture is supported by a number of tools providing a software compilation and execution environment.

3.3.1 Jamaica Assembler and C Compiler

A toolset comprising a C compiler, based on the Princeton LCC compiler [50] and an assembler is used in order to generate binary boot images for the Jamaica

architecture. The C compiler is able to compile a sizeable subset of the C language directly to the Jamaica ISA, but support for multithreading is not available, and so where required as in the boot procedure, small hand coded Jamaica assembly routines supplement the C generated code.

3.3.2 Jamaica Boot Procedure

The Jamaica architecture, in simulation, implements a cold start protocol whereby only a single context, the *primordial* context, begins the execution of code, all other contexts start in the *empty* state[4]. Prior to execution, the simulation environment loads an ELF binary, containing a boot procedure, into physical memory placing code and data segments at addresses detailed by the ELF file. The start address is extracted from the ELF file and is used as the initial PC value for the *primordial* context.

The code contained in the boot procedure is responsible for initialising registers and memory, including the initialisation of interrupt vectors and loading any other required code into physical memory. After this initial phase all *auxiliary* contexts can be woken using a software interrupt, `SIRQ`. The software interrupt vectors execution into an initial wake-up routine that sets up a minimal stack for each context capable of handling code shipped via the `THJ/THB` instructions. Upon completion of this phase each context releases a token onto the work distribution ring and switches to the *idle* state awaiting incoming work.

3.3.3 The Jamaica Virtual Machine

Software execution is supported on the Jamaica architecture primarily by the Jamaica Virtual Machine (JaVM) [40], a port of the Jikes Research Virtual Machine (RVM) [1] to the Jamaica instruction set. The Jikes RVM compiles and optimises Java bytecode to native machine code. The JaVM port allows execution of unmodified Java applications on top of the Jamaica architecture.

[4]Section 3.1.1 discusses the states that contexts can reside in.

JaVM Boot Procedure

Supplementary to the standard Jamaica boot procedure, boot strapping JaVM ensures that the *primordial* context and all *auxiliary* contexts are associated with a `VM_Processor` object, and that key Java class files are loaded into memory. The `VM_Processor` object maintains a set of queues containing Java threads associated with it. These threads are run within the context that the `VM_Processor` is attached to, illustrated in Figure 3.14.

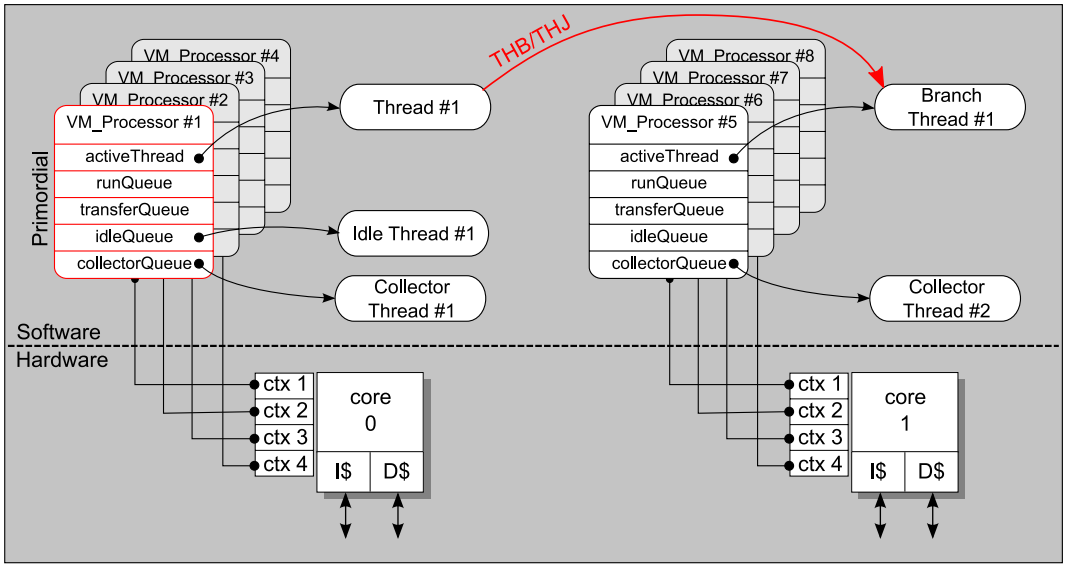


Figure 3.14: *JikesRVM software to Jamaica hardware mapping: Each hardware context is associated with a* `VM_Processor` *object.*

Idling Contexts

After the initial boot strapping phase, each context is associated with a `VM_Processor` object. The `VM_Processor` is responsible for accepting and scheduling any Java thread created by the virtual machine or application code for execution on the context. In the Jikes RVM a `VM_IdleThread` resides inside the `idleQueue` of a `VM_Processor`. The `idleThread`, a small loop checking for new threads in the `runQueue`, is run whenever the `runQueue` becomes empty. In the JaVM the `idleThread` immediately schedules a `VM_BranchThread`, a thread that sets up a small stack to handle incoming threads distributed across the shared bus using the `THJ` operation. The `VM_BranchThread` then exits from the bottom of its

working stack, releasing a token onto the work distribution ring, and freezing the hardware context, and therefore the associated `VM_Processor`, in the *idle* state.

Work Distribution

When a thread resident on another `VM_Processor` creates a new `VM_Thread` object, which encapsulates ordinary Java threads, it can attempt to locate a token from the ring using the `TRQ` operation. If it succeeds the `THJ` operation is invoked, supplying a schedule method as the restart address with the new thread as argument. The idle context immediately enters the *runnable* state and executes the method on the `VM_BranchThread` stack, inserting the new thread into the `runQueue` of the resumed context's `VM_Processor` and yielding until the new thread exits. When the branch thread finally resumes it exits, releasing another token. If a `VM_Processor` is not able to find an idle context to ship the new thread to, the thread is placed into the local `runQueue`.

Using these processes, regular multithreaded Java application code, and the threads within the virtual machine which is also written in Java, benefit from the work distribution mechanism provided by the Jamaica architecture. Additionally the `VM_BranchThread` is capable of executing an arbitrary shipped method, enabling it to be used to implement lightweight thread distribution.

3.4 Jamaica Simulation Environment

The previous sections of this chapter have discussed the Jamaica CMP architecture and its associated software environment. As Jamaica is a simulated architecture, this section describes the simulation platform developed as part of this thesis to study and extend the architecture, and to provide an environment for explorative software development.

3.4.1 Simulation Accuracy

Architectural simulations are typically a trade-off between speed and accuracy, with a complete spectrum ranging from circuit-level timing delay simulations [114]

through to cycle accurate and functional simulation [18], emulation and dynamic-binary translation [14]. The Jamaica simulation platform [67, 68], `jamsim`, is a Java simulation platform that has been developed to execute binaries created for the Jamaica instruction set. The `jamsim` platform supports several models of simulation. It can be used for fast, functional simulations required for system software development as well as cycle-level simulations, which are essential for quantitative evaluation of the architecture.

At cycle-level accuracy the simulation platform models the components in sufficient detail to account for effects such as stalls due to pipeline hazards, interconnection bus and queue contention, cache access contention and memory channel queueing.

3.4.2 Simulation Configuration

The `jamsim` simulation platform allows architectures to be configured based on the Jamaica instruction set. Parameterisable components of the simulated architecture, include the number of processing cores, the number of contexts per core, the L1, L2 and Level 3 (L3) cache sizes and ways, the size of the branch history table, the type of memory hierarchy and the interconnection network, either bus-based, crossbar based or a hybrid of both.

The simulation platform is capable of simulating the processor, the interconnect and the memory hierarchy both at the cycle-level and at a purely functional level. Where it makes sense the simulation platform can be composed of components at different levels of modelling. An example of this would be cache simulations, where it may not be necessary to use a cycle-level model for the processors as a functional model is able to generate the memory access patterns necessary to exercise the caches.

From scratch this simulator was developed as part of this thesis. The simulation platform is a structural simulator and currently consists of over 50 components, and some 30,000 lines of Java code. Each hardware component has been mapped onto a simulator component, a Java class, using object oriented practices. Additionally interfaces have been developed to enforce compatibility between multiple models of key components within the simulation platform, such as the processors,

```
private void initializeArchitecture() {
   CycleLevelProcessor[] proc = new CycleLevelProcessor[noProcs];
   L1CacheController[] iCache = new L1CacheController[noProcs];
   L1CacheController[] dCache = new L1CacheController[noProcs];

   for(int p = 0; p < noProcs; p++) {
      proc[p] = new CycleLevelProcessor(this, noCtxs);
      iCache[p] = new L1CacheController(this, L1size, L1sets);
      dCache[p] = new L1CacheController(this, L1size, L1sets);
      proc[p].connectCaches(iCache, dCache);
   }

   CacheController l2cache = new CacheController(this, L2size, L2sets);
   Bus bus = new Bus(this, l2cache, iCache, dCache);
   MemoryController memCont = new MemoryController(this, l2cache);
}
```

Figure 3.15: *Configuration code for building a CMP architecture in* ***jamsim****.*

caches, interconnects and memory controllers and to allow simple configuration. These interfaces have simplified the process of extending and adapting the current architecture models.

The simulation platform can be configured to run Jamaica instruction set binaries and Java class files through the JaVM port of the JikesRVM, targetting from single-threaded single-core systems, right through to hundreds of cores and multi-threaded, multi-cluster architectures.

3.4.3 System Simulation

As Jamaica is a simulated architecture and no device interfaces exist, see Section 3.1.7, complete system simulation is enabled using a special range of built-in instructions, refer to Appendix A.3. These instructions, which attempt a jump subroutine call, `JSR`, to small negative memory addresses are trapped during simulation, and the simulation platform calls out to the underlying operating system through the Java virtual machine in which the simulation is running, illustrated in Figure 3.17.

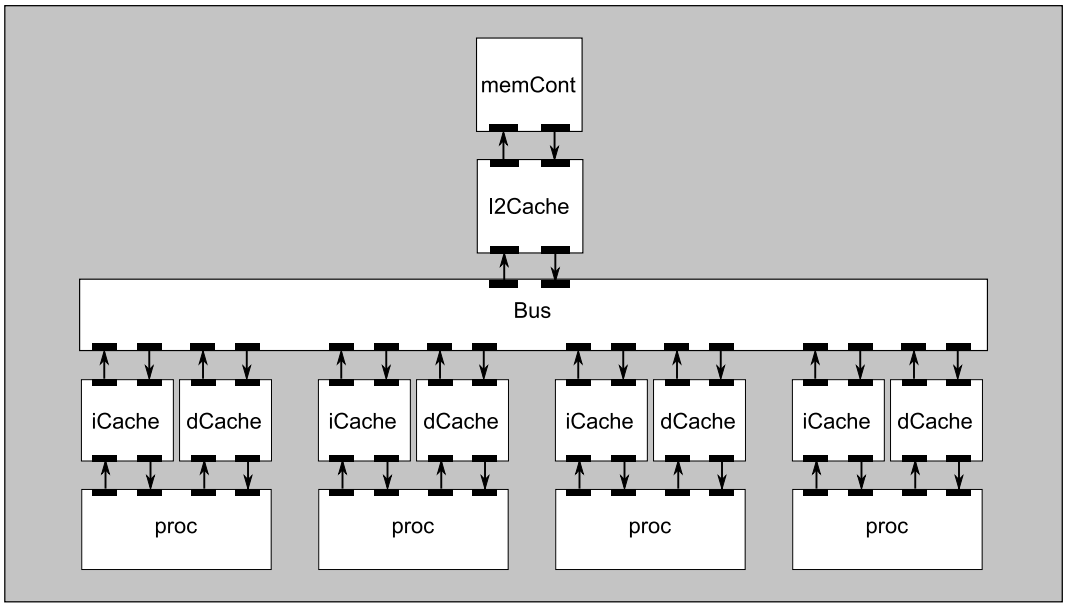


Figure 3.16: *Connected simulation components for* `jamsim`*, consistent with the configuration code listed in Figure 3.15.*

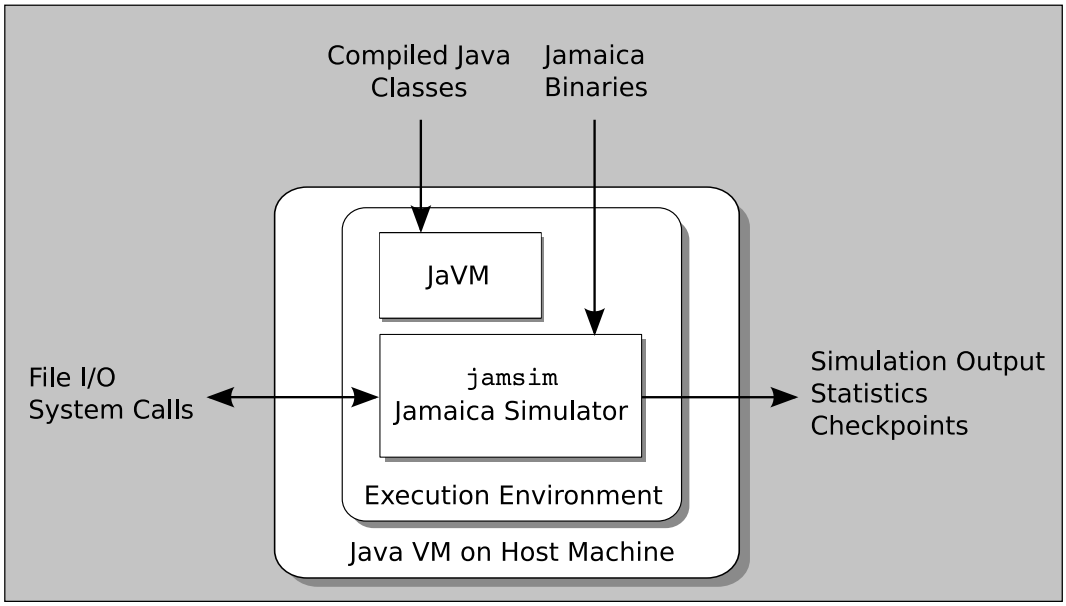


Figure 3.17: *Jamaica Simulation: Java bytecode is executed through the JaVM by* `jamsim` *within a Java virtual machine on top of the host system.*

3.5 Summary

This chapter has outlined the Jamaica CMP architecture as introduced by [170] and subsequent revisions made to it for this thesis. Each core contains a simple 5-stage RISC pipeline and accesses memory through private L1 instruction and data caches. Each core maintains sequential consistency and keeps coherent with other cores, a shared L2 cache and memory via a split transaction snoopy bus and a derivative of the MOESI cache coherence protocol. The architecture is supported by a collection of tools allowing both C and Java to execute on the simulated architecture. The `jamsim` simulation platform, developed as part of this thesis, was presented as a platform capable of simulating the Jamaica CMP architecture at both cycle-level and functional-level accuracies. In the following chapter an extension to the architecture is introduced that allows multiple CMP clusters to coexist within a chip, while still adhering to a shared memory paradigm.

CHAPTER 4

Multi-level Cache Coherence

As the number of transistors integrated within a single chip continues to grow the ability to increase the number of processing cores within a chip becomes possible. As more cores are added to a CMP architecture considerable pressure is placed on the memory hierarchy. Sufficient bandwidth is required to keep all of the cores working efficiently, and low latency is beneficial for inter-core communication.

This chapter briefly reviews alternative schemes for scaling up the memory hierarchy of CMP architectures. In the context of these schemes the limitations of the single shared bus Jamaica architecture are discussed and a novel multi-level cache coherence protocol is introduced which extends the memory hierarchy of the Jamaica architecture.

4.1 Multiprocessor Organisation

Much prior research within the multiprocessor field has looked at scaling architectures beyond tens of processors. This research established several categories of

multiprocessor organisation, with respect to both memory access [149, 117] and inter-processor communication [105].

4.1.1 Memory Access

Two categories of memory access in multiprocessor systems exist: distributed memory and Symmetric MultiProcessors (SMP). In SMP systems a single global shared memory is accessible to all of the processors within the system. The latency of memory access from each of the processors is uniform and as such SMP architectures are also referred to as Uniform Memory Access (UMA) architectures. Distributed memory systems, in contrast, usually have multiple memory modules each paired with one or more processors, as illustrated in Figure 4.1.

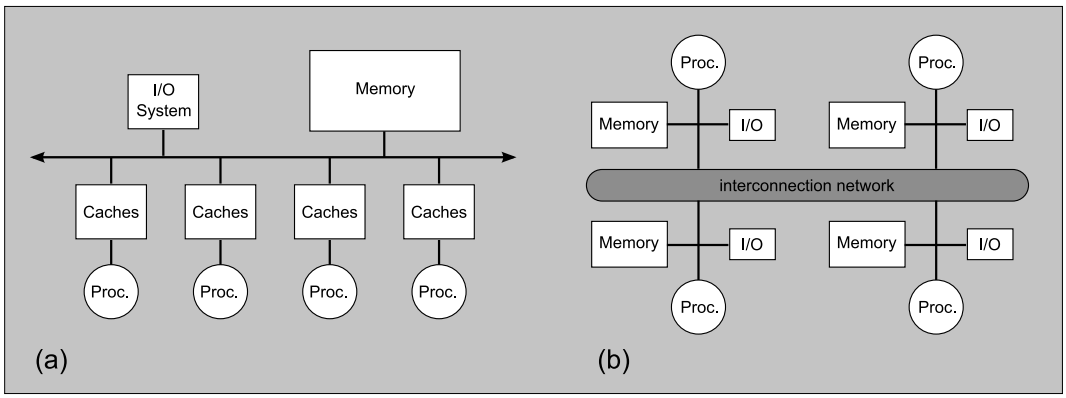


Figure 4.1: *Multiprocessor Memory Access: (a) Shared Memory, (b) Distributed Memory.*

Two variants of distributed memory systems exist. The first, *distributed shared memory* (DSM), also referred to as Non-Uniform Memory Access (NUMA) architectures, divides the global address space equally amongst the multiple memory modules [2, 97]. Access by any processor in the system to an address must be directed to the memory module containing that portion of the address space, usually controlled by a directory based coherence scheme [26]. A DSM multiprocessor from a programming perspective appears identical to a SMP, however the latency of memory accesses to local memory modules is far less than accesses to remote memory modules [80]. The second variant of distributed memory architectures divides the total address space into multiple private address spaces local

to each memory module [9]. These private spaces, which are disjoint, are not accessible by remote processors, in effect each processor-memory pair is essentially a separate computer.

In considering the scalability of a multiprocessor architecture, distributed systems appear to have two key advantages over SMPs. Firstly, if most memory accesses can be contained within the address range of the local memory module, the memory bandwidth of the architecture scales with the number of memory modules. The second advantage of a distributed memory arrangement is a reduced memory access latency. This lower latency again is realised if most accesses go direct to the local memory nodes. A disadvantage to a distributed approach is that additional software complexity is required to balance the memory access patterns made by each processor in order to utilise the bandwidth and latency benefits. This additional complexity increases the overhead associated with distributing work to the multiple processors in the system and limits the granularity of parallelism that can be exploited.

4.1.2 Inter-Processor Communication

In order for the processors within a multiprocessor system to speed-up the execution of an application, inter-processor communication is necessary to coordinate distribution of the overall workload and to synchronise on shared data. Two methods are employed for communicating data amongst the processors in a multiprocessor system. In shared memory multiprocessors, both SMP and DSM, communication occurs through the shared address space. Data is implicitly exchanged through load and store operations within the shared memory space, with every processor becoming aware of any changes through cache coherence [149]. In distributed memory systems where memory is disjoint and private to each processor, communication of data is achieved by explicitly passing messages amongst the processors [105].

A considerable disadvantage of the message-passing paradigm is that sharing of data must be explicitly annotated within software. This leads to a greater degree of software complexity [163] and again these additional overheads often reduce the amount of fine-grained thread-level parallelism that can be exploited [150].

4.2 Scaling the Jamaica Architecture

The Jamaica architecture [170], discussed in Chapter 3, is a CMP architecture consisting of multiple simple processing cores connected via private L1 caches to a shared bus which in turn is connected to a globally shared L2 cache and an on-chip memory controller. A limiting factor to this approach is the shared bus that connects all of the processing cores.

4.2.1 Limitations to Bus Scaling

As the transistor budgets of future process technologies increase the viability of incorporating more processing cores into the Jamaica architecture becomes realistic. However, the single-shared bus within the Jamaica architecture becomes a bottleneck to memory accesses as the number of cores is increased. Figure 4.2 shows the theoretical peak utilisation of the shared data bus in the Jamaica architecture, assuming high L1 instruction and data cache hit rates, 99% and 98% respectively, a perfect L2 cache hit rate and typical[1] RISC code [63], 22% loads and 12% stores. As illustrated, depending on the ratio of the bus frequency to the core frequency the bus begins to become a bottleneck, even for a relatively fast bus clocked at a quarter of the core frequency, the bus becomes saturated after 16 cores. Bus utilisation levels of around 80% have been shown to create detrimental increases in access delays largely due to queueing effects [168].

This problem is further exacerbated by wire delay limits, discussed previously in Section 1.2.1. If the number of cores connected to a single bus is increased then the bus will necessarily have to span further across the chip. As illustrated in Figure 1.1, in future process technologies as little as 10% of the die area will be accessible within a single clock. To put this into context, consider the parameters in Table 4.1.

If the Jamaica core architecture is considered to be of a similar complexity to the Alpha 21064 [106], then incorporating 64 cores onto a 65nm technology chip requires consideration of both wire-delay and bus scaling. Referring to Table 4.1, each core spans approximately 5.5% of the die length in the technology, and a

[1] SpecInt92 average instruction mix as reported in [63].

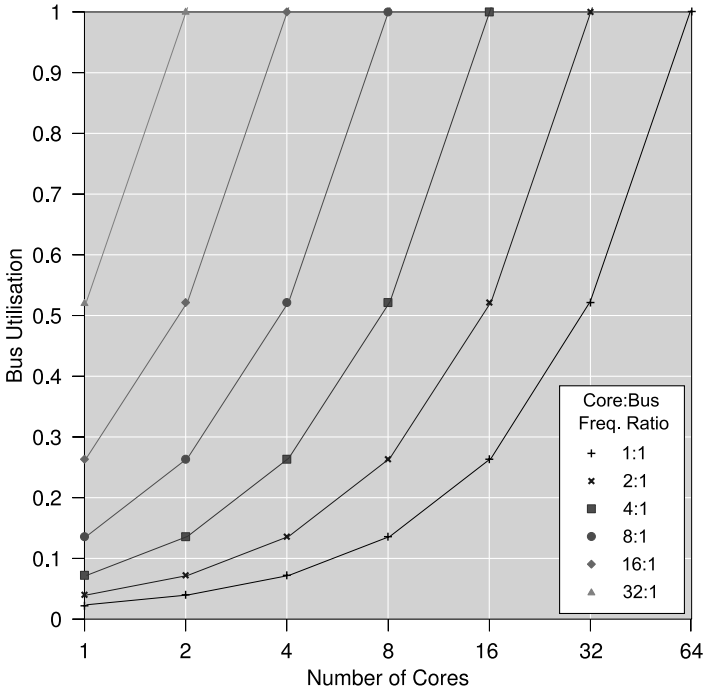


Figure 4.2: *Theoretical bus access limitations, assuming a 98% L1 data cache hit rate, a 99% L1 instruction cache rate, a typical RISC code mix [63] and a perfect L2 cache hit rate.*

65nm technology	1 billion transistors
14MB cache (6 transistors/bit)	704,643,072
64 (Alpha 21064) cores	179,200,000
Die span per core	approx. 5.5%
Die span 8-core bus	approx. 22%
Signal propagation 8-core bus	4 clocks
8 core utilisation at 4:1 (see Figure 4.2)	52%
Architecture	8 x 8 core clusters
128 × L1 caches (total 2MB)	each 16KB (I$ and D$ per core)
8 × L2 cluster caches (total 4MB)	each 512KB
1 × L3 cache (total 8MB)	each 8MB

Table 4.1: *A feasible configuration for scaling a CMP using 1 billion transistors.*

bus connecting all 64 cores, depending on the topology, would be required to span the length of some 30 core spans in order to connect them all, requiring a stretch some 165% of the die length. This would clearly lead to an infeasible design because the signal propagation on the bus would take more than 16 clocks (see Figure 1.1), and such a bus would be saturated by only 4 working cores (see Figure 4.2). A scalable memory hierarchy is therefore required to utilise the 1 billion available transistors fully, and as shown in Table 4.1, 8 clusters of 8 cores connected by such a hierarchy could provide a feasible design solution.

4.2.2 Multi-Level Cache Hierarchy

To increase the ability of the Jamaica architecture to scale with the addition of more processing cores the single shared bus architecture is replaced by a scalable multi-level cache hierarchy. The multi-level hierarchy, illustrated in Figure 4.3, maintains shared memory coherence, a pre-requisite for efficiently running standard multi-threaded applications written in high-level languages such as Java.

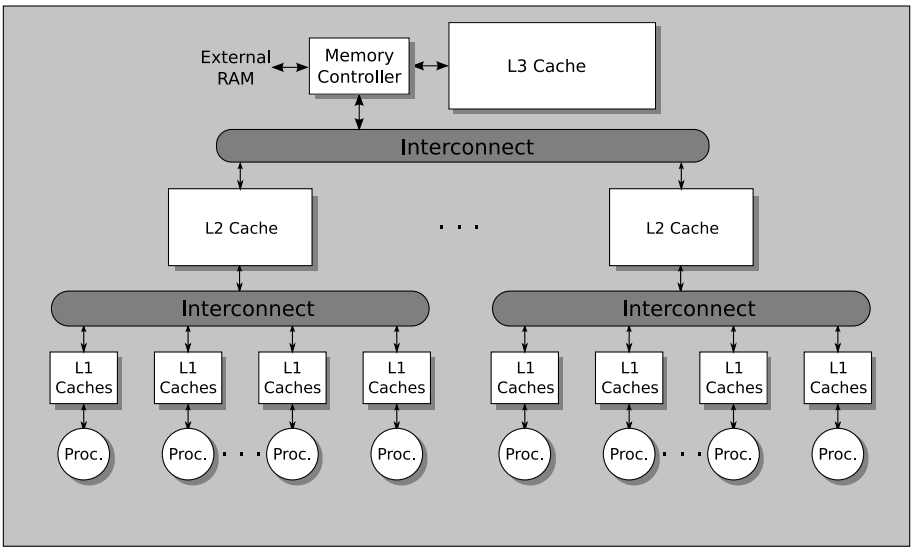


Figure 4.3: *Jamaica multi-level cache hierarchy.*

The multi-level hierarchy, by dividing the total number of cores into *clusters* each connected through a hierarchy of interconnect networks and caches, can allow many more cores to be integrated onto a single chip, whilst maintaining shared memory and limiting the span of each interconnect to reduce the effects of cross-chip wire delay and bus contention.

Each intra-cluster network is independently arbitrated and accessed concurrently allowing the cores within each cluster to access the larger cluster-shared cache with less contention. The additional scalability, however, comes at the expense of a more complex cache coherence protocol that needs to maintain coherence across multiple clusters, and the need to maintain cache inclusion.

A Chip Multi-Cluster (CMC) architecture, incorporating multiple on-chip clusters each containing multiple cores and multiple levels of shared cache is feasible given the transistor budgets of modern process technologies.

4.2.3 Cache Inclusion

The Jamaica memory hierarchy, as outlined in Section 3.1.5, allows L1 private caches to take ownership of cache lines avoiding inclusive L2 caches. Once ownership for a cache line is passed onto an L1 cache, the line containing the non-owned copy in the L2 cache is redundant and can be freed. This removes the necessity for the L2 cache to include the set of all lines contained within the L1 caches which potentially allows the L1 and L2 caches, when combined, to contain more data.

In a multi-level hierarchy inclusion is important for shielding intra-cluster networks from the traffic of inter-cluster networks at each level [11]. Without inclusion a multi-level cache hierarchy has no way of shielding inter-cluster coherence messages from the intra-cluster networks, and an unnecessarily large amount of traffic is generated.

Non-Inclusive Write Request

As an example consider a write request, illustrated by Figure 4.4. When a write request to address A misses in the L1 cache (L1$[4]) attached to core 4 (P[4]), a request is forwarded onto the L2 cache (L2$[2]). Without inclusion, the request made to L2$[2] must be visible to other L1 caches serviced by L2$[2], as they may hold a copy of the data at address A. In a hierarchy of buses the write request placed onto the intra-network bus L1N[2] would be snooped by L1$[5] allowing response of the data or an acknowledgement that the data is not present. In a network hierarchy an explicit invalidation message would need to be sent directly

to L1$[5] and acknowledged with a message before the write request is propagated to the next level in the hierarchy.

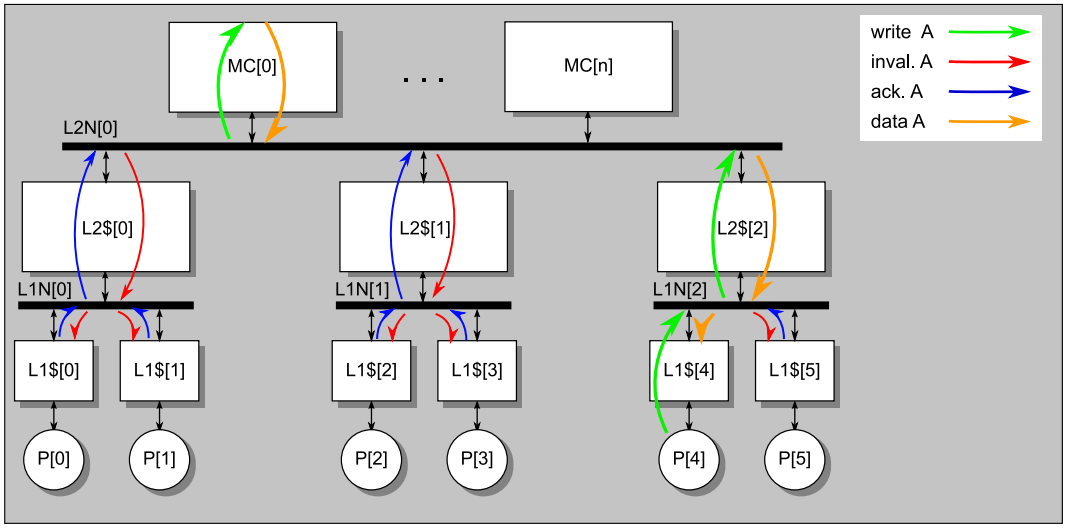


Figure 4.4: *In the absence of inclusion coherence messages must be forwarded to each and every cache creating unnecessary network traffic, and acknowledgement of each invalidation must be received before propagating requests to successive levels.*

In the example the data at address A is not present in any cache, so the request made to L2$[2] is forwarded onto the inter-cluster network (L2N[0]). Again before the request can be forwarded to the memory controller (MC[0]), invalidations must be made visible to caches L2$[0] and L2$[1], in turn all caches below them, L1$[0]-L1$[3] must receive and acknowledge an invalidation message. Finally when the invalidations have successfully been sent and acknowledged by all caches the request can be forwarded to the memory controller and a response including the data at address A can be returned to L1$[4] which allows P[4] to continue execution.

Inclusive Write Request

By maintaining inclusion, the same request from core 4 would only generate traffic on the networks containing copies of the data at address A. If the data is not present in any cache in the system, the network traffic is reduced to the propagation of the request and response, as shown in Figure 4.5. Where inclusion information exists, messages do not need to be sent to lower level caches. In the example, cache L1$[5] does not need to be sent an invalidation message as L2$[2]

knows that no copies exist in any lower level caches. This is not the case on the inter-cluster network L2N[0] as the memory controllers contain no inclusion information, and so messages must be sent to both L2$[0] and L2$[1] before forwarding the request to MC[0]. The invalidations sent to L2$[0] and L2$[1] are simply acknowledged, but no change of state is necessary as they do not contain the data at address A, and no messages are forwarded to caches L1$[0]-L1$[3].

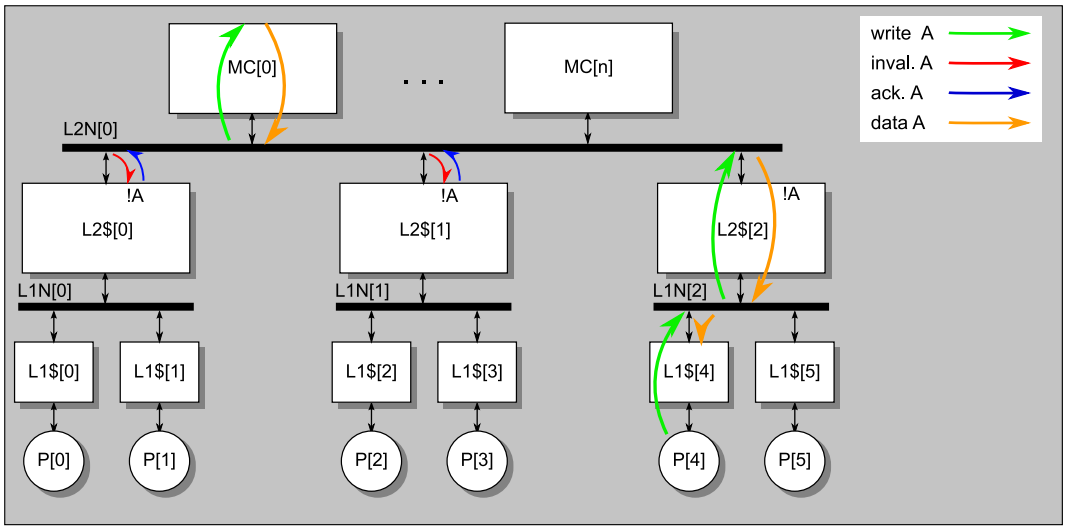


Figure 4.5: *Maintaining inclusion reduces any unnecessary traffic being generated in clusters not containing copies of the requested data.*

Maintaining Cache Inclusion

A disadvantage of maintaining cache inclusion is that the set of lines in each shared cache must be a superset of all of the cache lines contained within the caches sharing it. To avoid poor hit rates in shared caches, they need to be significantly larger than the sum of all the caches connected below them [168]. The space overhead of inclusion can be reduced by allowing certain lines, for example those modified in lower level caches, to be cleared from the shared cache, if a table of address tags for these lines is maintained and accessed in parallel to the main cache tags [13].

4.2.4 Locality and Affinity

A further aid to scalability in a multi-level cache hierarchy is the implicit exploitation of locality. The multi-level hierarchy exploits both spatial locality and parallel locality. Spatial locality is exploited in the same manner as all cache architectures, each cache line fetched contains multiple words. Memory references made in the near future have a high probability of being near recent past references, and therefore multiple references may be made to the same line, removing the necessity for multiple memory requests.

Parallel locality is an extension of the effect of spatial locality in the context of a parallel program. Future memory accesses by a thread can be predicted by recent memory access patterns of related threads, in the same parallel program, in addition to its own recent accesses.

Significant levels of spatial and parallel locality are usually present in parallel programs [49]. Parallel locality can be increased by explicitly enforcing an affinity onto threads, distributing them in such a manner as to keep related threads within a subset of the cache hierarchy. This same process can be used to insulate applications, as much as possible, from interference by unrelated threads.

4.3 PIMMS - a Multi-Level Coherence Protocol

Having outlined in previous sections the motivation for extending the Jamaica architecture to allow a scalable multi-level shared cache hierarchy, this section introduces the PIMMS[2] protocol, which maintains system wide cache coherence. The protocol maintains compatibility with the original Jamaica instruction set and as such code written for that architecture can run unmodified on the multi-level hierarchy.

Unless otherwise stated, the examples presented in this chapter assume a hierarchy of buses connecting the shared level caches. The protocol presented is, with minor modifications, capable of maintain coherency additionally across a hierarchy of crossbars.

[2]PIMMS is an acronym for the 4-bits used to encode all states; Pending, Invalid, Modified, Modified Stale.

State	Code	Description
Invalid	I	no line present
Valid	V	read access only
Valid Shared	V*	as Valid, shared by lower level cache(s)
Modified	M	read and write access
Modified Shared	M*	as Modified, shared by lower level cache(s)
Modified Stale	MS	line stale, modified by lower cache
Pending	P	operation pending, refuse access

Table 4.2: *PIMMS protocol: cache states.*

4.3.1 Cache States

The protocol used is extended from the family of MOESI protocols [153] with additional states to allow multi-level cache hierarchies. Ownership is discarded as it is implied by maintaining inclusion. The protocol is similar to those used in the KSR-1 [49], Paradigm [29] and Gigamax [168] multiprocessors. Any cache line can be in one of seven states listed in Table 4.2, except for lines in private L1 caches which can only be in the states I, V, or M. Only lines within the L1 data cache can reside in the Modified state. A cache line in the Modified Stale state additionally keeps track of the index number of the cache in the lower level that currently holds the modified copy. Although similar to the protocol states presented by Anderson and Baer [6] for multi-level hierarchies, the seven states and index tracking maintained by the PIMMS protocol reduce unnecessary coherence messages within the system and allow the protocol to generalise to non-broadcast networks.

It should be noted that the states V* and M*, where the star denotes the line as being shared by lower level caches, are weak annotations in a bus based hierarchy. When a sharer exists the Sharer state is always set, however the state may remain set even after a sharing cache has overwritten the shared line, and therefore no longer shares that line. Infrequently this leads to invalidation messages being sent to a cache no longer containing a copy of the line; these messages are ignored.

Where crossbars are used to connect cache levels in a hierarchy the Sharer states, V* and M*, must maintain a list of sharers. This list of sharers is used to determine the channels within the crossbar that must be reserved in order to send an invalidate signal.

Class	Code	Name	Description
memory bound	SH	Share	Request for read access to a line
	MD	Modified	Request for write access to a line
	MC	Cond. Modified	As Modified but from a STL_C
	WB	Writeback	Writeback/Eviction of a modified line
core bound	MSH	Mem Shared	Response with read access and data
	MMD	Mem Modified	Response with write access and data
	INV	Invalidate	Force invalidation of line
	DWN	Downgrade	Force downgrade of line (e.g. M → V)
	MWB	Mem Writeback	Force writeback of line

Table 4.3: *PIMMS protocol: network transactions, mnemonic codes and descriptions.*

4.3.2 Network Transactions

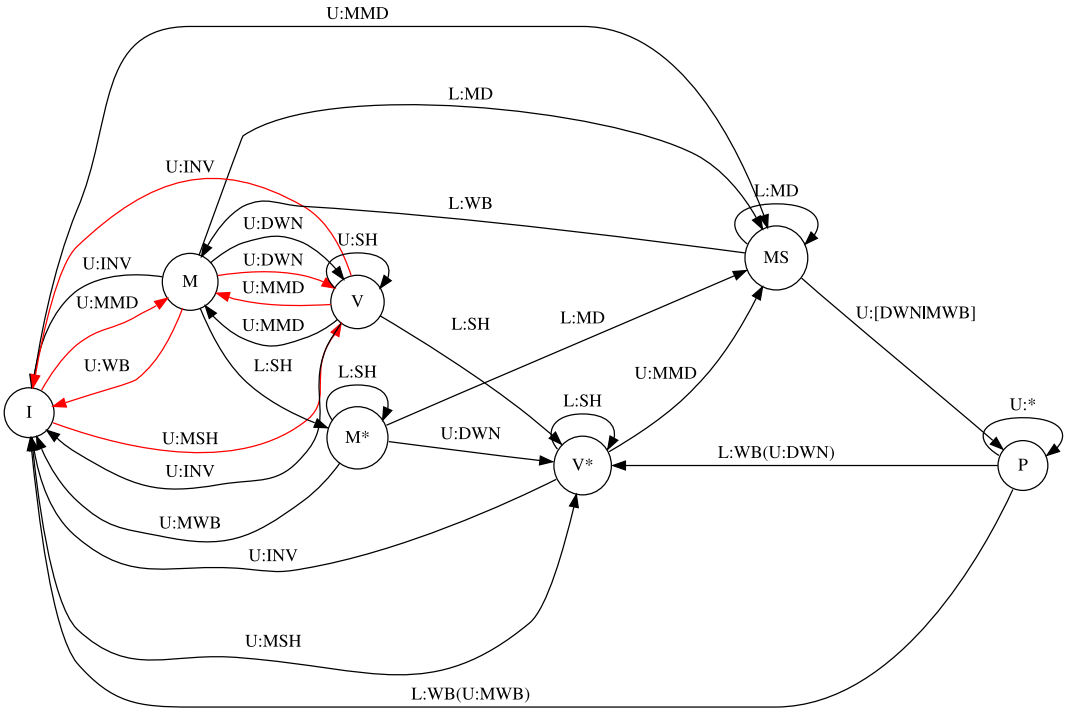
Two classes of transactions are generated in the protocol. Those originating from a cache on the core side of an interconnect that propagate in the direction of memory are referred to as memory-bound transactions. Transactions originating from the memory side of the interconnect, propagating towards the core side, are referred to as core-bound transactions. In total eight types of network transactions exist, listed in Table 4.3.

4.3.3 State Transitions

Figure 4.6 shows all of the possible transitions between the seven cache states, when network transactions occur on the upper or lower interconnects surrounding a cache. The possible state transitions in the L1 caches are far fewer as they only include the states V, I and M.

4.3.4 Four Phase Transactions

In a single shared bus architecture a request placed on the bus either receives a response from another cache holding a copy, or after a delay from memory. In both cases all caches that either hold a copy or require a copy can alter state after snooping a transaction for the same data on the bus. In a multi-level cache hierarchy, however, data can be present in caches that are not directly connected



KEY	
U:	upper network
L:	lower network
SH	request read access
MD	request write access
WB	writeback/evict line
MSH	response read access + data
MMD	response write access + data
INV	force line invalidation
DWN	force downgrade line
MWB	force writeback line
[A|B]	A or B
(A)	following A
*	any transaction seen
→	L1 cache transitions

Figure 4.6: *Multi-level cache state transitions for shared level caches. Note that for clarity MC transactions are left off the diagram as apart from their handling internally in the cache controller queues, see Section 5.6, the state transitions are identical to MD.*

to the same interconnect as the requesting cache. As a result two additional scenarios are encountered within the hierarchy:

1. Multiple transactions for the same data can be generated on separate interconnects concurrently.
2. Copies of data may be modified in caches not shared by the requesting cache.

The implication of the first scenario is that two requests that have started can meet at a shared interconnect both competing for the same data. This is handled in much the same way as with a single bus architecture; the requests are handled in sequence, after the first transaction is completed the second may progress. The second scenario requires an extension from the two phase (request, response) bus transactions employed in single shared bus architectures, to four phase transactions (request, action, reaction, response).

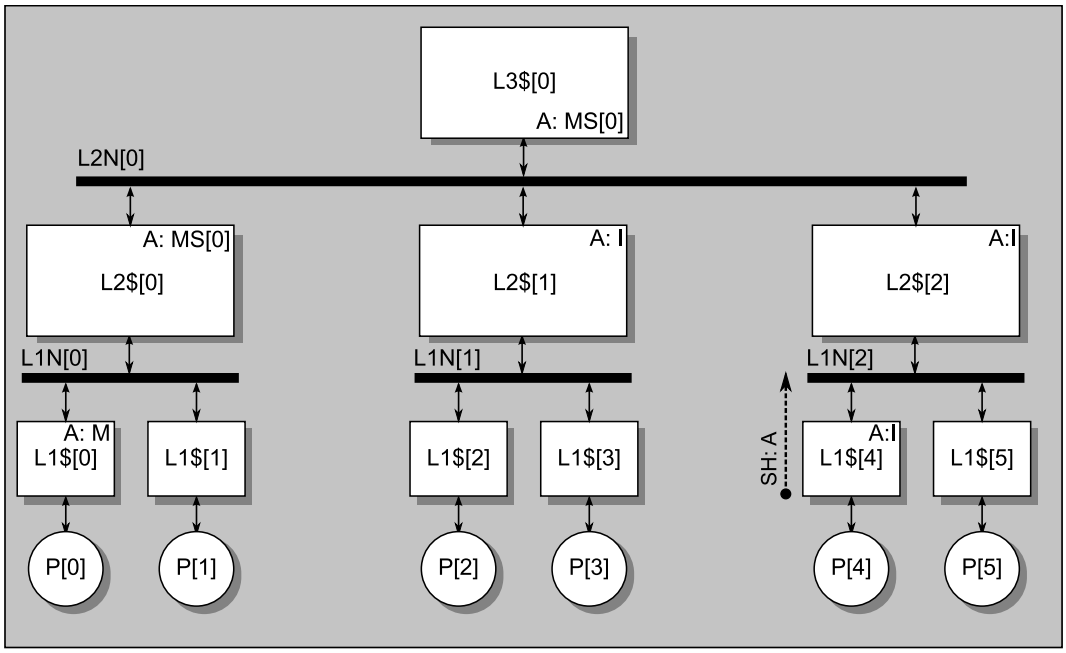


Figure 4.7: *Four phase read transaction.*

Four Phase Read Transaction

As an example of a four phase transaction consider the scenario illustrated in Figure 4.7. Core P[4] issues a read for data at address A, the most upto date copy of which resides in L1$[0]. On issuing a read to the interconnect L1N[2] a miss is triggered in the cache L2$[2]. L2$[2] forwards the read request and issues it on the interconnect L2N[0]. The cache L3$[0] currently holds the line in state MS. The modified stale state indicates that the line is present in the cluster under cache L2$[0], and none of the caches connected to the interconnect L2N[0] can supply an up to date copy of the data because the data has also been modified by a cache below L2$[0]. In order to be able to respond to the read request made by P[4], the modified data must first be fetched from L1$[0]. An *action*, in this case downgrade (DWN), is issued to L2$[0] and eventually to L1$[0]. L1$[0] downgrades the line and issues a *reaction*, in this case a writeback (WB) along with the data. When the WB and data are issued on the L2N[0] interconnect, a response can be forwarded to the original requestor, L1$[4]. A timing diagram of the four phase read transaction, including cache line state transitions, is shown in Figure 4.8.

Four Phase Concurrent Write

The potential for two concurrent requests for the same data to arrive at a shared bus concurrently is illustrated in Figure 4.9. Both core P[4] and P[0] are attempting to gain write privileges to data at address A. Both private L1 data caches, L1$[0] and L1$[4], only contain the data in the valid state and therefore propagate MD requests up the hierarchy to the L2N[0] network in order to acquire the line in a modified state from the L3$[0]. Assuming both requests arrive at the interconnect L2N[0] concurrently, one of the requests is given priority, in this case the request originating from L1$[0].

As outlined in the timing diagram, Figure 4.10, as the MD request from L1$[0] is responded to with a MMD, an invalidate is triggered in the L2$[2] cluster to invalidate its copy of the data. After the initial MD request has cleared from the interconnect the second MD request can be issued. During this phase the second MD request triggers a MWB to fetch the data from the cluster beneath cache L2$[0]. It is possible that the original MMD is still propagating towards L1$[0],

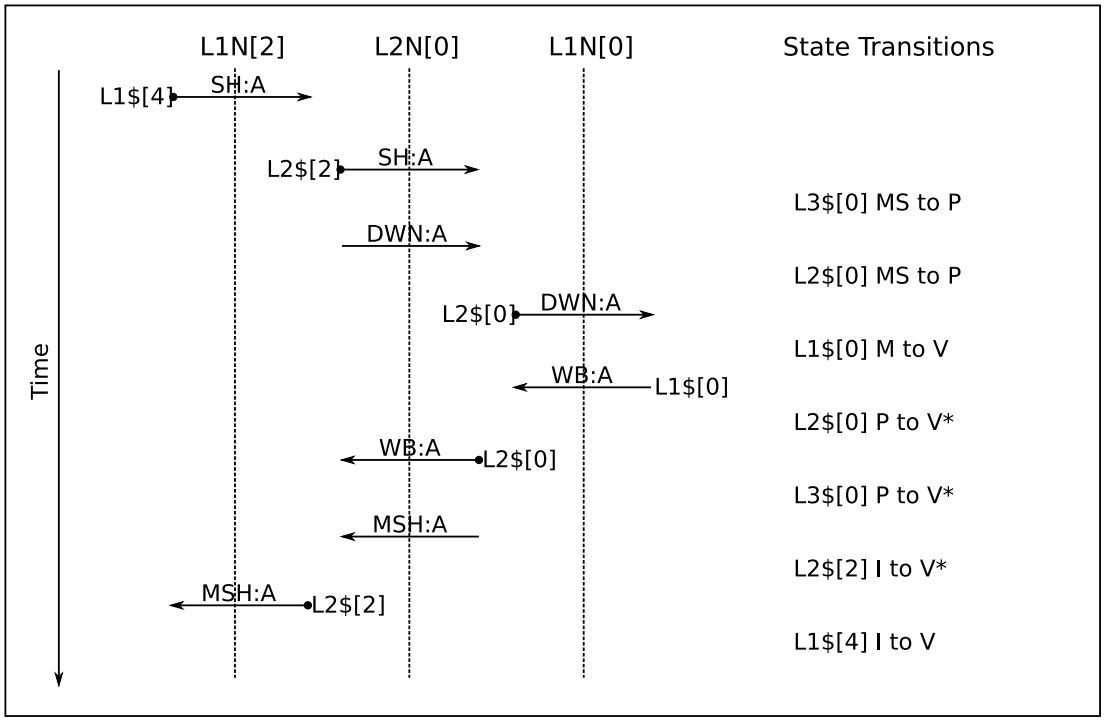


Figure 4.8: *Four phase read transaction, timeline.*

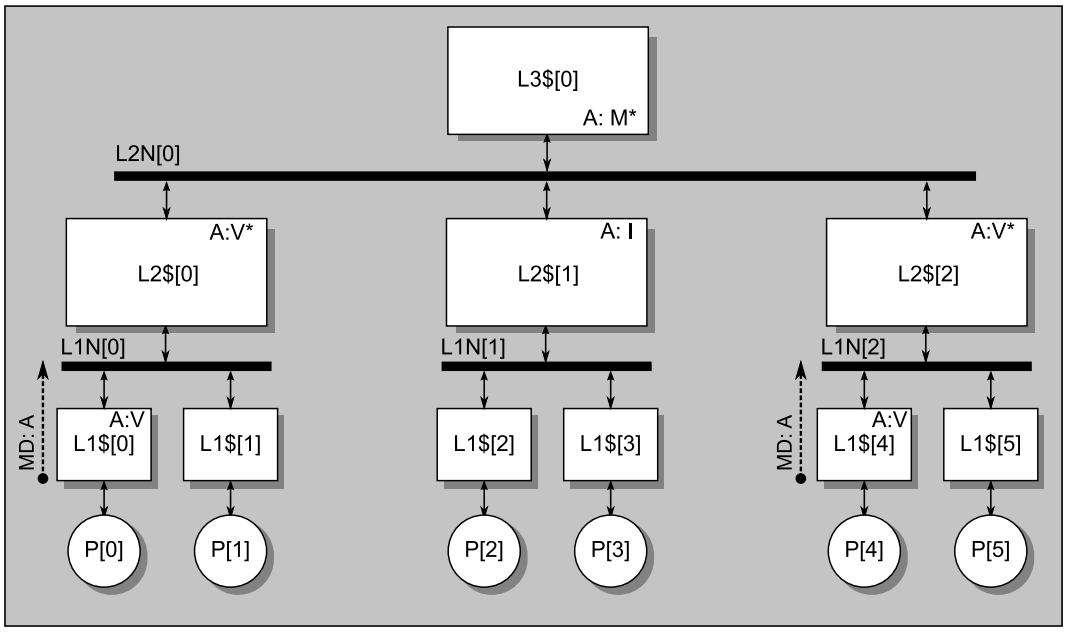


Figure 4.9: *Four phase concurrent write transactions.*

and the network must ensure ordering so that the original request is responded to and has time to commit the write before the data is written back in response to the MWB. Following the writeback the data is sent with modified permissions to the cache L1$[4]. As can be seen concurrently writing to the same location from cores in separate clusters generates considerable network activity.

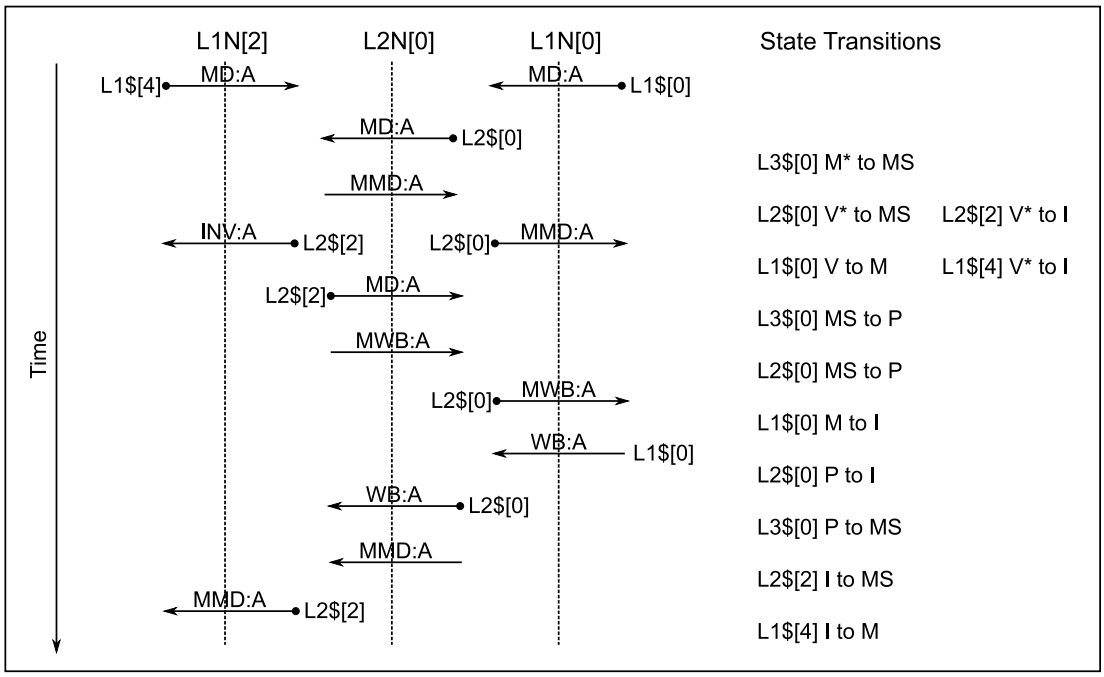


Figure 4.10: *Four phase concurrent write transaction, timeline.*

Pending State

A potential difficulty with the four phase transaction is that while an *action* is in progress, within a cluster, another transaction outside of the cluster may request the data involved. In the read example, from the point at which the downgrade (DWN) is issued until the point that the line is written back (WB), the state of line A is in flux. If the state of line A in L3$[0] is left as modified stale (MS) during this period a read request from a cache above the L3$[0] would cause the L3$[0] to issue a downgrade (DWN) onto the L2N[0] interconnect. This downgrade would be issued into the L2N[0] network. This would not only cause unnecessary network traffic, but has the potential to generate a downgrade to a cache that has already downgraded the line. To prevent occurrences of such transactions, a

line is set in the Pending state (P) while an *action* is in progress. Any request to the line is negatively acknowledged and must be retried after regaining access to the network.

4.4 Summary

This chapter has outlined the scaling limitations of the single shared bus design currently employed by Jamaica and several other CMP architectures [60, 108, 80, 83]. In order to fully increasing transistor budgets fully the integration of many more cores on-chip will be desirable. At the same time maintaining a shared memory hierarchy simplifies the implementation of parallel applications and allows exploitation of finer grained parallelism.

Furthermore, this chapter introduced the concept of a CMC architecture and a protocol based on four phase transactions, that is able to keep a CMCs multi-level cache hierarchy coherent. A novel aspect of the protocol is the Pending state which prevents unnecessary inter-cluster traffic entering a cluster while the data requested is being altered by another four phase transaction.

In the next chapter the hardware support required to implement the multi-level PIMMS coherence protocol is introduced.

CHAPTER 5

Multi-level Cache Hardware

The introduction of a multi-level cache hierarchy into the Jamaica CMP architecture requires significant changes to the cache hardware. Moving from a single shared bus to a hierarchy of buses or other interconnects introduces networking issues which must be handled by the caches. Multithreaded cores and the addition of the four-phase coherence protocol allow many outstanding transactions to be in transit in the multi-level hierarchy at once. The cache hardware must be able to support these transactions, prevent the hierarchy from becoming easily saturated and avoid deadlock. This chapter describes the cache hardware, used to implement the coherence protocol described in the previous chapter. The cache hardware has been implemented, through detailed simulation, for a hierarchy of buses, crossbar switches and a hybrid of both.

5.1 Cache Organisation

The multi-level cache hierarchy consists of two types of cache, the private L1 data caches and a number of shared caches depending on the depth of the hierarchy. The organisation of both types of cache is outlined in the following sections.

5.1.1 Level 1 Private Caches

As mentioned in Section 3.1.5 the L1 caches are shared by multiple contexts within each of the processing cores. Each context is stalled on both an instruction cache and data cache miss, restricting the number of outstanding operations in the cache hierarchy to two per context. This restriction also ensures that the memory accesses made by each context remain sequentially consistent [87], despite any reordering that may occur higher up in the cache hierarchy.

The L1 caches in the multi-level hierarchy, shown in Figure 5.1, differ little from those in the single shared bus architecture. When a memory operation misses in the L1 cache an entry is made into the cache request table which holds one request per context. On gaining access to the interconnect, the interconnect side controller issues a request from the request table or writes back a line from the writeback buffer. Writeback buffer entries are given priority over entries in the request table.

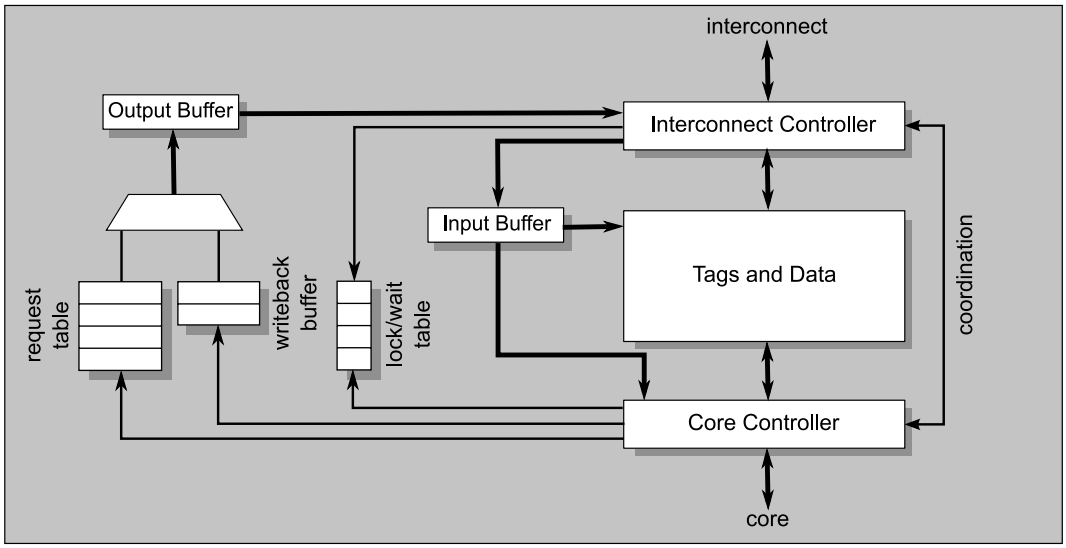


Figure 5.1: *Level 1 cache.*

Responses, invalidations or writeback requests originating from higher level caches in the hierarchy are also handled by the interconnect controller. As both the core and the interconnect controllers share a single access port to the tag and data array in the L1 caches, contention between the two can occur and is resolved by giving priority to the interconnect controller. Should queue congestion occur, which is possible in the data cache, the interconnect controller is able to block or

negatively acknowledge incoming writeback requests until a slot in the writeback buffer is freed.

5.1.2 Shared Level Caches

The second type of caches in the multi-level hierarchy are shared level caches. Shared level caches are generally shared by multiple lower-level caches. Although this is not a strict requirement, they are generally larger than the sum of all the caches directly sharing them, and are necessarily slower to access. Importantly the shared level caches act as a bridge between two levels in the hierarchy and are connected to two interconnects, see Figure 5.2.

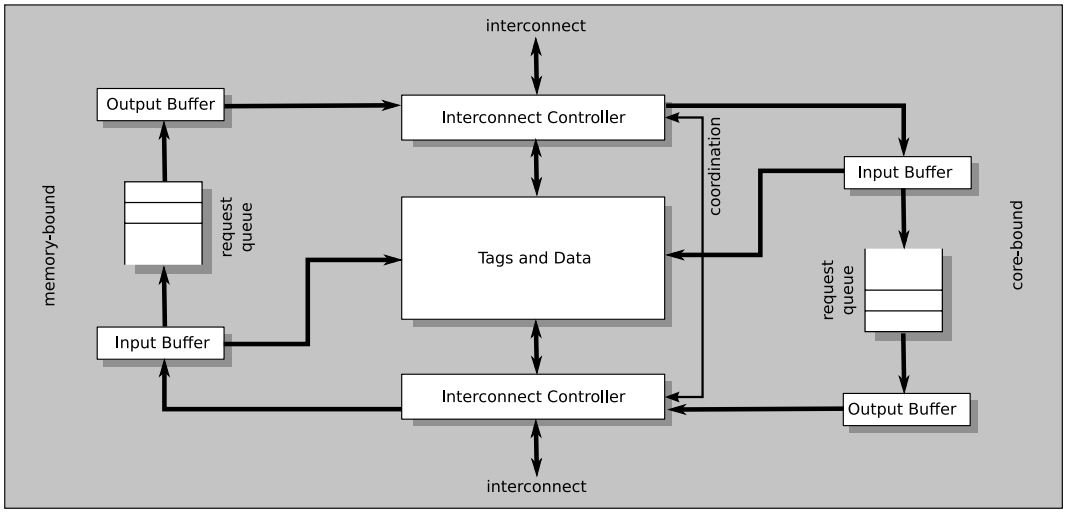


Figure 5.2: *Shared level cache.*

As mentioned previously, in Section 4.3.2, two classes of network transactions exist, memory-bound transactions and core-bound transactions. The shared level cache, in bridging two interconnects, is required to accept and process memory-bound and core-bound transactions, update the cache tags and data, and forward the requests or responses to either higher or lower level caches. A separate queueing channel is implemented for each class of transaction, and a separate controller is required for the lower and upper interconnects. Again access to the cache tags is shared by both the upper and lower interconnect controller, with the upper controller having priority when contention arises.

5.2 Coherence Messages and Transactions

In the following discussions of flow control and deadlock avoidance within a multi-level cache hierarchy the terms *coherence message* and *transaction* are used.

A *coherence message* is used to mean the actual physical message sent through the hierarchy. In the multi-level hierarchy this coherence message consists of the request or response type, the address and optionally the data. The memory- and core-bound request queues, illustrated in Figure 5.2, are required to store coherence messages.

A *transaction* simply refers to the process of delivering a *coherence message* across the interconnect in a multi-level hierarchy. An incoming transaction, for example, refers to a coherence message being sent across the interconnect and arriving at a cache's interconnect controller.

5.3 Flow Control

In the multi-level cache hierarchy requests and responses may potentially propagate through multiple interconnects and caches; requiring buffering into queues at each level. These queues can overflow due to interconnect saturation, and to prevent the loss of transactions a mechanism is required which guarantees message delivery. Additionally, in the process of maintaining coherency, a single incoming transaction can trigger multiple outgoing transactions; for example, upgrading a line in one cache and concurrently invalidating all sharers of the same line. In such cases the incoming transaction cannot be removed from a buffer until all of the outgoing transactions have been delivered.

To handle flow control in the multi-level cache hierarchy, each shared level cache is encapsulated by a core-bound and memory-bound FIFO based queue system, see Figure 5.2. Incoming coherence messages are buffered into the input buffer and then processed by the interconnect controllers which also maintain the coherence protocol logic. As previously mentioned, access conflicts by the interconnect controllers to the cache tags are resolved by giving priority to the higher level interconnect controller, the one closer to memory and generally running at a lower frequency.

The interconnect controllers are also responsible for forwarding coherence messages between levels when required by copying the messages from the input buffers into the request queues. When an incoming transaction simply requires that the shared level cache changes state, for example a writeback meeting a modified stale line, the coherence message is consumed by the interconnect controller and is not forwarded. When a transaction triggers responses or actions the interconnect controller is responsible for generating and issuing them.

5.3.1 Blocking and Negative Acknowledgments

When an incoming transaction requires that the shared level cache generates a coherence message, for either a response or action, there is a possibility that the message may be blocked by the destination cache. This can occur when the destination cache is no longer accepting incoming transactions as the relevant request queue has reached or is approaching capacity.

When such an event occurs the original incoming transaction is either negatively acknowledged (Nack'd), where the interconnect is a bus, or the transaction is left within the buffer slot it occupies in a crossbar fabric. In both cases the transaction is subsequently rescheduled. Rescheduling occurs until the transaction is able to complete because the relevant destination cache's request queue is no longer blocked. Simulations done for this thesis have shown that queue blocking does occur frequently in larger many core architectures.

In order to avoid deadlock, particularly when the memory-side cache controller has a blocked transaction, and to avoid rescheduling repeatedly blocked core-side caches an exponentially increasing, 7-bit saturating block counter is used. Each cache controller only arbitrates for the interconnect when the block counter is 0. If a transaction is nack'd then the block counter is incremented in powers of 2, until it saturates at 128. The block counter is decremented by 1 for each missed arbitration slot, until reaching zero and retrying the transaction.

5.4 Deadlock Avoidance

In multi-level cache hierarchies, in particular in the interconnect and queues that connect them, deadlock can occur. This is because all four conditions required for deadlock to occur [33] are present. In particular deadlock can arise in multi-level hierarchies between the memory-bound and core-bound queues leading to a circular chain of dependencies [109], an example scenario is shown in Figure 5.3. Both the write request (MD A) in the memory bound queue and the writeback request (MWB B) in the core bound queue are blocked as they both generate responses, which cannot be buffered in the cache's core and memory bound queues and deadlock arises.

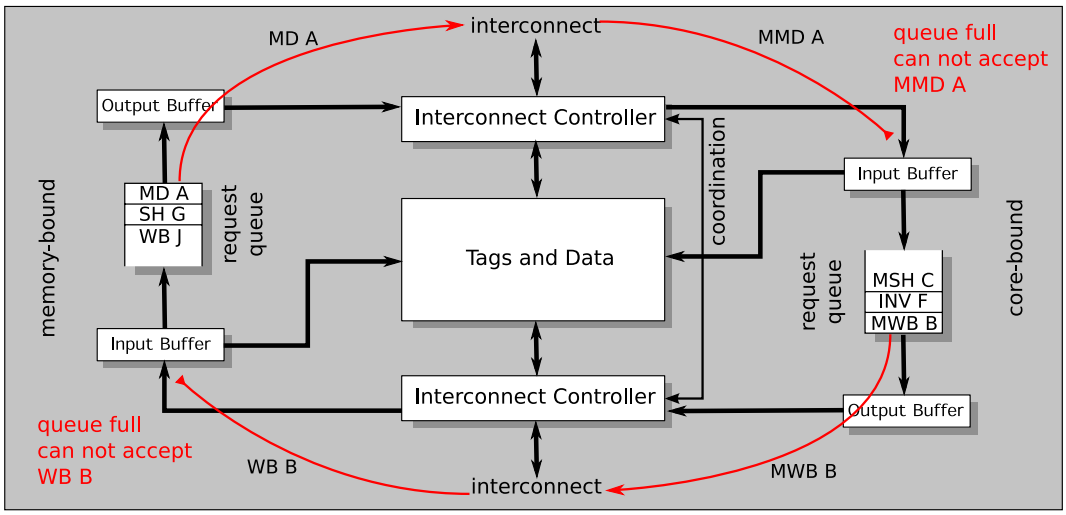


Figure 5.3: *A circular dependence between the queues leading to deadlock.*

To overcome deadlocks arising from circular dependencies the queueing system within the shared level caches is extended in a similar manner to the NUMAchine architecture [101, 55]. The request queue is divided into two separate physical queues handling different classes of coherence messages. For queueing purposes two classes of coherence messages exist.

5.4.1 Sinkable Messages

Sinkable messages are coherence messages that do not elicit a response back into the interconnect. In the PIMMS protocol these messages include writebacks

(WB) and invalidations (INV). Neither of these messages generate additional coherence messages back into the network which generated them. The messages are either consumed or forwarded by a shared level cache.

5.4.2 Non-Sinkable Messages

Non-sinkable messages are coherence messages that do elicit responses back into the interconnect. In the PIMMS protocol these messages include read requests (SH), write requests (MD and MC), downgrade requests (DWN), writeback requests (MWB) and additionally read and write responses (MSH and MMD). The responses are included in the list of non-sinkable coherence messages as, due to the lazy allocation of cache lines, a response can evict a modified line generating a writeback.

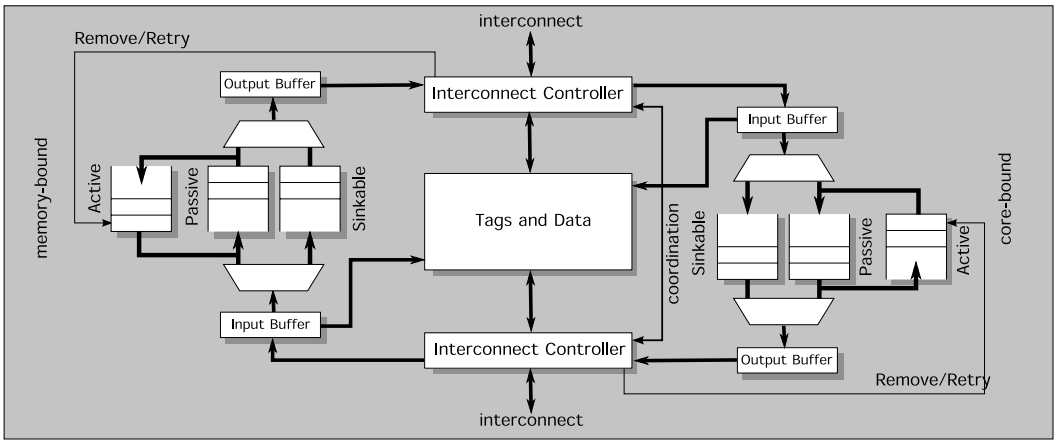


Figure 5.4: *Shared cache request queues divided into sinkable and non-sinkable entities. Non-sinkable queue divided further into passive and active queues allowing reordering.*

5.4.3 Sinkable and Non-Sinkable Queues and Priorities

The queueing structure resulting from splitting the input queues into sinkable and non-sinkable queues is shown in Figure 5.4. The dependencies leading to deadlock in the previous scenario, illustrated in Figure 5.3, are now avoided. The writeback (WB B), generated by the core bound memory writeback request (MWB B), is now guaranteed to find space in the memory-bound sinkable queue.

This in turn frees a slot in the core-bound non-sinkable queue which then allows the write request (MD A) to be issued.

In general deadlock is avoided by ensuring three rules are adhered to by all of the coherence messages in the multi-level hierarchy:

1. sinkable messages remain ordered,
2. sinkable messages are guaranteed to propagate, and
3. sinkable messages are always given priority over non-sinkable messages.

5.4.4 Passive and Active Non-Sinkable Messages

Unlike the scheme developed by Grindley *et al.* [55] for the NUMAchine architecture, the queue containing non-sinkable coherence messages is additionally split into two further queues, a *passive* and an *active* queue, again refer to Figure 5.4. The passive queue is used to hold coherence messages prior to gaining access to the interconnect network for issuing. The active queue is used to maintain a copy of non-sinkable coherence messages currently in the process of being issued across the interconnect. When a coherence message has been issued the entry is removed from the active queue. If during the issuing the interconnect controller determines that a non-sinkable coherence message can not complete, then the entry is removed from the active queue and inserted back into the passive queue. Upon reaching the head of the passive FIFO the coherence message is retried.

Allowing non-sinkable messages to become re-ordered in the passive and active queues allows the processing of coherence messages to be pipelined and prevents a single message from unnecessarily blocking the progress of other messages in the queue. Each message, in the simulated implementation, can only be re-ordered twice, at which point the message gains blocking priority over the non-sinkable passive queue. Limiting re-ordering ensures that eventually each message will make forward progress.

Coherence Message Reordering

Figure 5.5 illustrates the function of passive and active queues when connected to a bus-based interconnect. Five snapshots of the memory-bound input queues for a shared level cache are shown. For clarity it is assumed that no other cache is competing for the bus during the period shown. The non-sinkable and sinkable queues contain three read requests and a single writeback request respectively. Following the deadlock avoidance rules, mentioned in Section 5.4.3, the writeback takes precedence over the three read requests and is issued first. At cycle 3, the writeback transaction is placed on the bus, freeing the non-sinkable queues to arbitrate for access to the bus. At cycle 5 the first read request is issued on the bus for address B, and a copy of the coherence message is placed in the active queue. At cycle 7 the second read request, this time for address C, is issued. A copy is placed in the active queue, and the data for the writeback is transferred on the data bus. During the same cycle, however, the upper level cache triggers a Nack for address B, signalling a blocked queue. The interconnect controller raises the retry signal, and the coherence message (SH B) is re-entered into the passive queue. At cycle 9 the third read request is issued. No Nack is triggered during this cycle so the copy of the coherence message for the second read request is cleared from the queue. Finally during cycle 11 the initial read request (SH B) is re-issued.

As the queue was blocked during the issue of the first read request the final ordering of the requests sent across the network is WB A, SH C, SH D and SH B. Were the three read requests being sent to a multiple banked cache, where each address was contained within separate banks, the blocking of address B would not significantly delay the progress made by the other requests. Re-ordering is also beneficial when a shared cache is issuing core-bound transactions to multiple cores or clusters. A coherence message being sent to a cluster or core that is blocking incoming transactions will not unnecessarily delay subsequent messages to other cores or clusters.

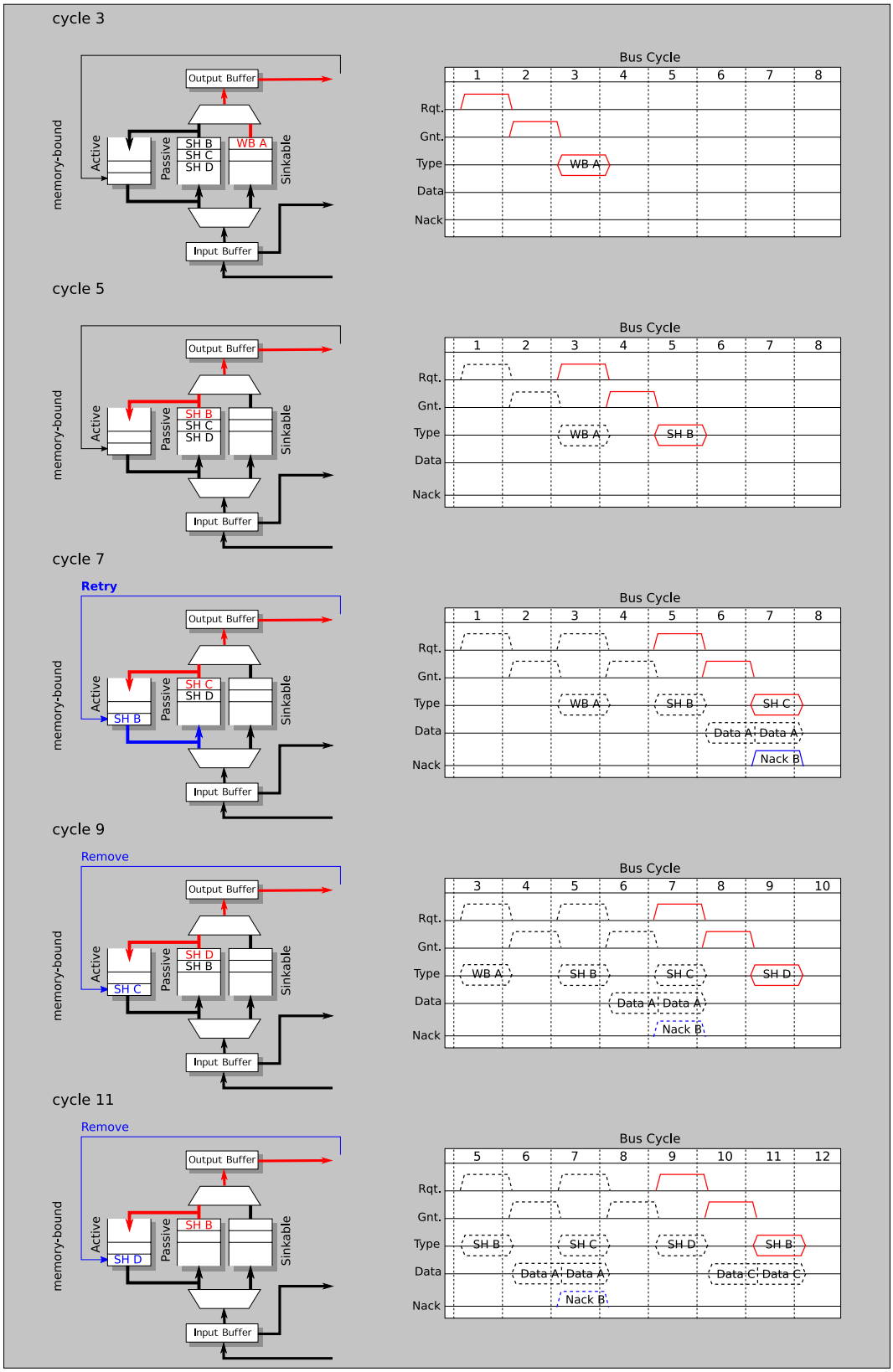


Figure 5.5: *Passive/Active queue reordering.*

5.5 Address Blocking

To reduce the number of coherence messages propagating through the multi-level hierarchy, and to simplify the protocol logic, each interconnect in the hierarchy implements address blocking. A small table of addresses is stored at the lower interconnect controller within each shared level cache, as shown in Figure 5.6. When a coherence message is propagated up to a higher level in the cache hierarchy the address is stored in the address blocking table. Should another coherence message for the same address arrive at the lower level interconnect controller it is blocked, and rescheduled as previously described in Section 5.3.1. The address blocking table also prevents multiple transactions for the same address from entering the same shared level cache concurrently. When the address blocking table is full, subsequent transactions that require propagation to higher levels are blocked.

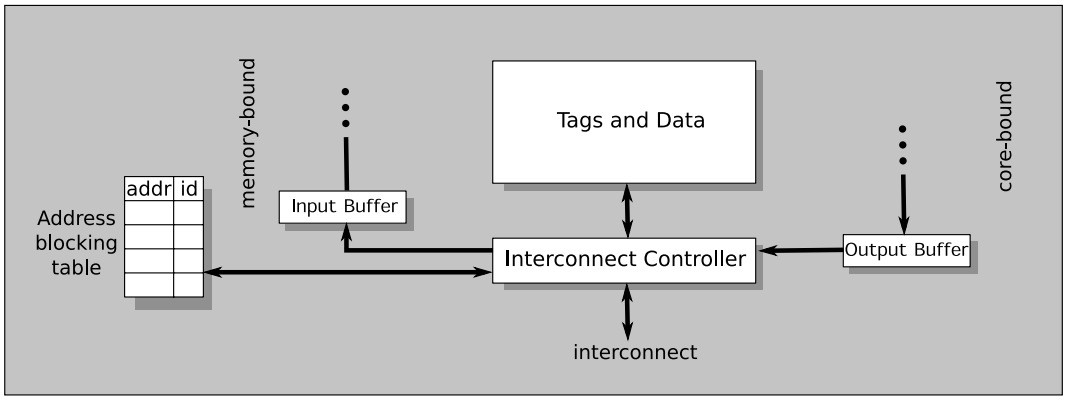


Figure 5.6: *Multi-level address blocking table.*

The address blocking table also stores the identifier of the cache that issued the transaction onto the interconnect. This id is subsequently used when a response is generated in order to route the coherence messages back through the hierarchy.

5.5.1 Local Transactions

The address blocking table is implemented as a separate structure to the cache tags and as such the size is necessarily limited to allow fast access. The table can therefore become full if many coherence messages all for different addresses are

sent to the upper levels of the hierarchy. During periods of significant activity locally contained transactions, those not requiring any further propagation, are still allowed access to the shared level cache. Local transactions may complete if no coherence messages are triggered to higher levels in the hierarchy by a change of cache state.

If the number of entries in the address blocking table is fewer than the total number of core side caches attached to the network, a portion of the address blocking table is reserved. This reserved section ensures throughput from local transactions is always maintained. When a local transaction completes its entry is removed from the address blocking table.

5.5.2 Deadlock Avoidance

The address blocking table is important for routing responses back to the original requesting caches, however it also introduces another possible deadlock scenario. Should the address blocking table block all coherence messages then it can easily become deadlocked, as illustrated in Figure 5.7.

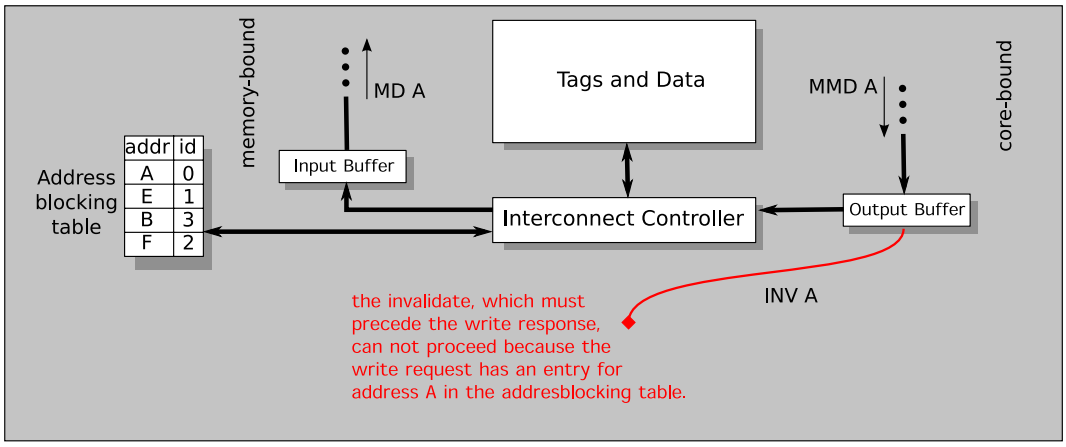


Figure 5.7: *Multi-level deadlock arising in the address blocking table.*

By applying the same rules outlined in Section 5.4, in particular guaranteeing the propagation of sinkable messages by not blocking them, deadlock can be avoided. Sinkable messages share the reserved portion of the blocking table with local transactions so that they can always propagate across the interconnect.

5.6 Multi-Level Synchronisation

Extending the cache hierarchy to multiple shared levels has implications when providing synchronisation instructions using the load-linked and store-conditional pair. Multiple processors can execute a load-linked instruction and subsequently attempt a store-conditional instruction, illustrated in Figure 5.8.

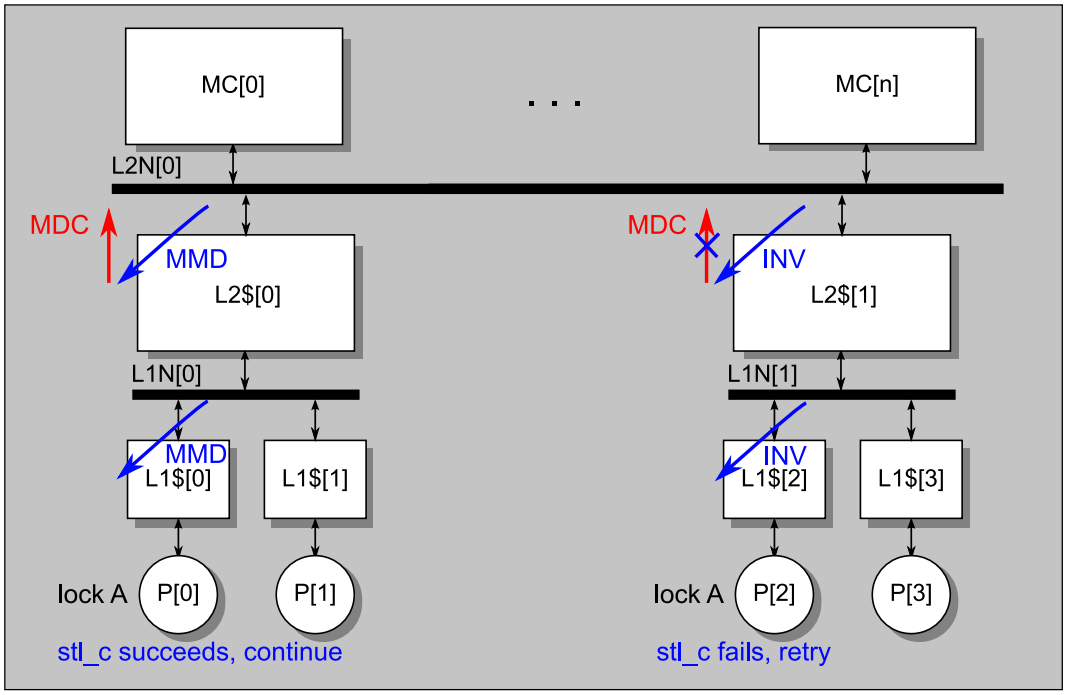


Figure 5.8: *Load-linked and store-conditional synchronisation in a multi-cluster architecture.*

The store-conditional instruction generates a store-conditional, MC bus transaction, which can be in flight for multiple processors concurrently. Arbitration for the top level bus will ensure that only one of the MC transaction succeeds, processor `P[0]` in Figure 5.8. A MMD response is generated towards processor `P[0]`, and because `P[2]`'s cache holds the line shared, an INV transaction is generated. The invalidate transaction is also required to remove any in-flight store-conditional transactions for the same address from all memory bound buffers that the invalidate signal passes. When the invalidate signal reaches the cache attached to `P[2]` the context waiting for the store-conditional to complete is woken and the store-conditional fails. Software routines, such as the one outlined in Figure 3.10, are responsible for a subsequent reattempt to enter the critical section of code.

This cancellation process is necessary to stop a subsequent MC generating an INV response which could reach the processor before the store response[1]. In such a scenario both store-conditional transactions would arrive at the processors with both sets of locks reset, and so would fail and need to be reattempted. Such a scenario can lead to livelock.

In the presence of multiple levels of shared cache it may be more appropriate to use the compare and swap primitive for certain concurrent algorithms to avoid the potential for *ping-ponging* between multiple load-linked/store-conditional pairs.

5.7 Lazy Cache-Line Allocation

Both the private and shared level caches within the multi-level hierarchy implement a write-back policy, and can therefore hold the only up to date copy of a given cache line. Write-back caches are advantageous in multi-level hierarchies as they generate less write traffic when compared to write-through caches [76], with only evictions generating transactions on the interconnect.

A lazy cache line allocation policy is implemented within the multi-level hierarchy. When a cache miss occurs a coherence message is generated and propagated into the network without allocating a line to hold the response. This occurs at each successive level in the hierarchy and no line is allocated until a response transaction is received.

When a response is received by a cache, the least recently used set for each line mapping is selected as an allocation candidate for the incoming cache line. If the candidate line is Valid the line is simply overwritten, if the line is Valid Shared the line is overwritten and invalidations are generated for the sharing caches. If the candidate line is Modified a writeback is generated, if it is Modified Shared a writeback and invalidations are generated. If the candidate is Modified Stale a writeback request is generated to fetch the latest data into the cache and the line is set Pending, however the response is delayed and must be rescheduled. The response is rescheduled until the candidate line is no longer in the Pending state and the line in the Modified state can be written back and overwritten.

[1]INV and WB are examples of sinkable transactions which are able to overtake other transactions due to the split-channels described in Section 5.4.3.

This policy ensures that when a line is finally allocated, after the response is received back at each level, the least recently used set in each cache is selected for eviction. Additionally during the period from a request being generated to a response being received no lines within the cache are reserved, which would reduce cache utilisation. Furthermore, lazy allocation prevents requests from being blocked when a particular mapping in the cache has been allocated across all of the sets for outstanding transactions.

5.8 Summary

This chapter has presented the cache hardware required to implement the cache coherence protocol presented in Chapter 4. The hardware implementation of shared level caches within the hierarchy was discussed, in particular the core- and memory-bound queueing systems were outlined. The addition of multiple levels of shared cache introduces networking issues such as flow control and deadlock.

Deadlock within the multi-level hierarchy is prevented by dividing the coherence messages into sinkable and non-sinkable messages and providing separate queueing channels for each. A novel passive and active queueing mechanism was presented that allows reordering of non-sinkable messages and prevents head of queue blocking. Finally lazy cache line allocation was introduced.

In the next chapter distribution and optimisation schemes are discussed that best utilise the multi-cluster architecture.

CHAPTER 6

Multi-level Task Locality

The extension of the Jamaica CMP architecture to a CMC architecture presents several challenges and opportunities for software applications. The ability to integrate many more cores within a single chip shared-memory architecture, potentially allows for greater performance but also increases inter-processor communication. If clustering is a possible direction for next generation CMP architectures then both application restructuring and scheduling to take advantage of locality of reference must be carefully considered.

This chapter discusses locality within the CMC architecture outlined in Chapters 4 and 5, presents a novel extension to the work distribution mechanism and the instruction set and discusses the use of this scheme to implement both application restructuring and scheduling.

6.1 Clusters and Cache Locality

In a multi-level cache hierarchy multiple levels of sharing exist; the contexts within a core share the L1 cache, all cores within a cluster share the L2 cache,

and all L2-clusters share the L3 cache or memory, as illustrated in Figure 6.1. This sharing continues to extend in deeper hierarchies as more cache or index[1] levels are added.

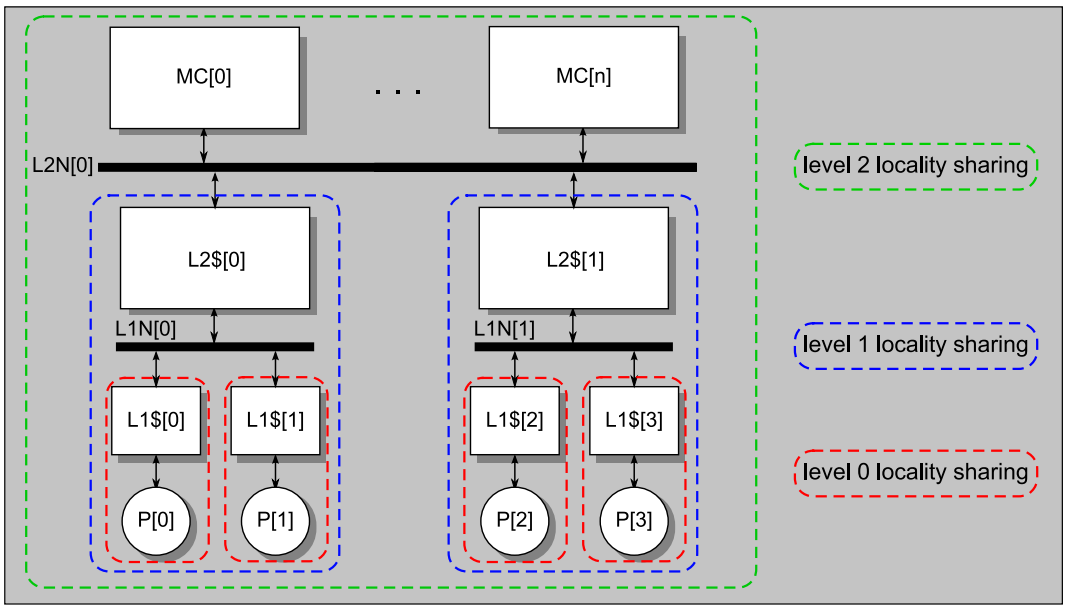


Figure 6.1: *The multi-cache hierarchy implicitly exhibits a hierarchy of locality of reference within each shared level working cache set.*

Ideally application tasks or threads should be distributed across the processing cores in an attempt to best exploit the locality between associated tasks. Balancing tasks in this manner reduces the bandwith requirements of higher level interconnects, reduces the visible memory latency, and avoids unnecessary congestion within the network.

6.2 Task Distribution

The Jamaica architecture, outlined in Chapter 3, provides hardware support for fine-grained parallelism by means of a token distribution ring, outlined in Section 3.1.3. The ring allows tasks to be distributed from a running thread maintained within one hardware context to any other idle hardware context within the CMP.

[1]Maintaining a cache at every level in the hierarchy is not strictly necessary, higher levels within the system can be coordinated using tag matrices [49], to reduce the transistor requirements.

Given a single bus CMP architecture the distribution of tasks is arbitrary and each idle context is given an equal weighting as a candidate for task distribution. This scheme works well as all contexts within a single bus CMP share a single L2 cache and any L1 cache can transfer data to any other L1 cache, because all shared data is accessed across the single shared bus. Some advantage can be gained from distributing two associated tasks to two idle contexts within the same processing core, as there will be some benefit from the sharing of data within a L1 cache, but can also be a disadvantage when another free context resides in a wholly idle core somewhere else on the chip.

6.2.1 Locality Aware Task Distribution

Extending the architecture to a multi-cluster CMP extends the access possibilities for data shared between multiple contexts. Data may be shared, and subsequently modified, by a context within a different cluster. Successive modifications to the same data by two contexts in two separate clusters will incur significant delays due to the latency of continually passing updated copies of the data across higher level interconnects and invalidating and moving the data through multiple levels of cache, as illustrated in Figure 6.2.

Minimising the level at which data is shared by associated tasks can significantly reduce the access latency to shared data by those tasks and, as a consequence, improve the performance of a parallel application.

Synchronisation also benefits from locating coordinated tasks within a cluster where possible. This ensures that data regions used to implement atomic primitives remain as close to the context attempting synchronisation as possible, minimising the latency of each synchronisation.

6.2.2 Token Requests

In the original single shared bus Jamaica architecture a token-request instruction, `TRQ`, is executed during the process of forking a task. The token-request either returns an integer value, uniquely associated with an idle-context somewhere on the chip, or after polling unsuccessfully for a software set number of cycles,

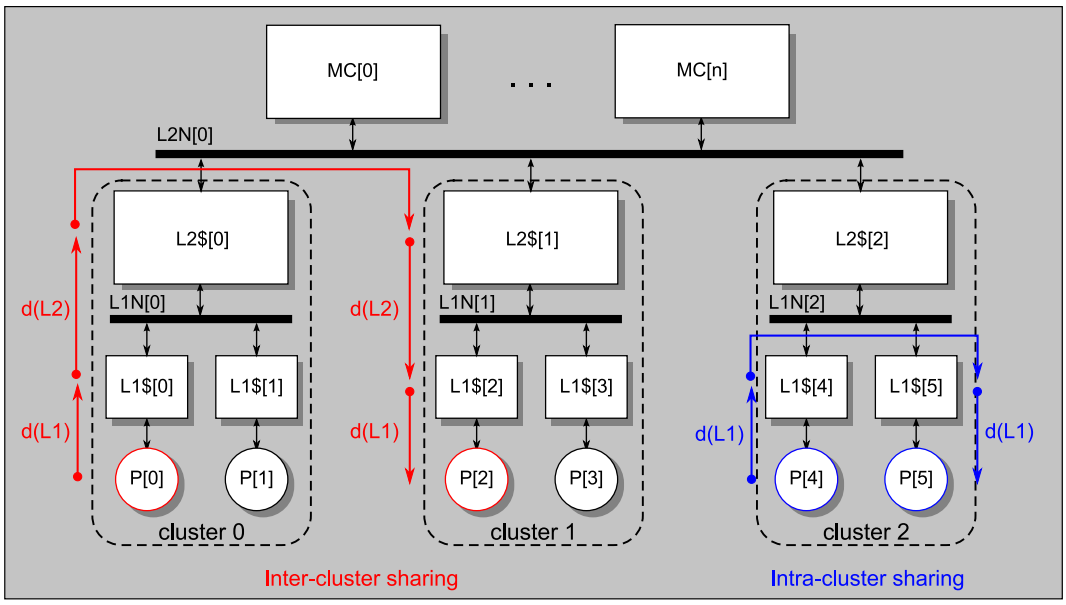


Figure 6.2: *Two contexts operating on the same data perform more efficiently if that data can be kept within the same cluster, intra-cluster sharing, as opposed to sharing between clusters, inter-cluster sharing.*

returns 0 informing the context executing the token-request that no idle context was available during the polling period.

When an idle context identifier is returned by the execution of a token-request, task setup data is sent across the single shared bus along with the idle context's unique identifier using the thread jump, `THJ`, instruction. The idle context, as part of constant normal cache snooping, recognises the identifier and reads the task data, eight 32-bit values, off the data bus and into the context's *In* register window[2], before beginning execution of the forked task. Part of the task data includes a value uniquely identifying the parent thread, which is required when notification is later sent back to the parent thread informing it that one of its forked tasks has completed.

In order to maintain this lightweight thread-shipping mechanism within a multi-cluster architecture the token-request instruction has been extended to encode information about the cache locality of each context.

[2]Register windows are discussed in Section 3.1.2.

6.2.3 Locality Aware Token Request Extensions

The CMC architecture exhibits a number of levels of cache locality, shown in Figure 6.1. Two contexts within a CMC architecture can either cohabit a cluster or exist in separate clusters at each level of cache locality. As an example all contexts within the processing core `P[0]`, in Figure 6.1, cohabit a level 0 cluster. A context in `P[0]` and a context in `P[1]` cohabit a level 1 cluster, but exist in different level 0 clusters. The minimum locality level at which two contexts share a common cache will be referred to as the *cache-distance* between them.

The cache-distance can be used to distribute tasks to contexts either within a locality level, or outside of a locality level. In order to calculate the cache-distance between two contexts, the token distribution mechanism is extended such that cluster information is encoded into the unique-identifiers.

6.2.4 Cache-Distance Identifiers

The CMC architecture and coherence protocol, introduced in Chapters 4 and 5, allow for an arbitrary configuration of the multi-cluster hierarchy. A CMC architecture can be configured in a balanced tree-like topology, as shown in Figures 6.1 and 6.2, or in an unbalanced topology, as shown in Figure 6.3.

By assuming that an architect may want to build or analyse the performance of both balanced and unbalanced cluster configurations the cache-distance identifier is encoded with sufficient information to determine the minimum level at which two contexts share data by comparison with another context's cache-distance identifier.

Algorithm 1 Cache-distance encoding.

1: *cache-distance id* = 0
2: **for** *l* = *top sharing level* down-to 0 **do**
3: components = max number of sharing components at level *l*
4: bit_shift = bits needed to express the number of components
5: bit_mask = index of the component the context is connected under at level *l* ***or*** index of component itself ***or*** 0
6: *cache-distance id* = *cache-distance id* | bit_mask
7: *cache-distance id* = *cache-distance id* << bit_shift
8: **end for**

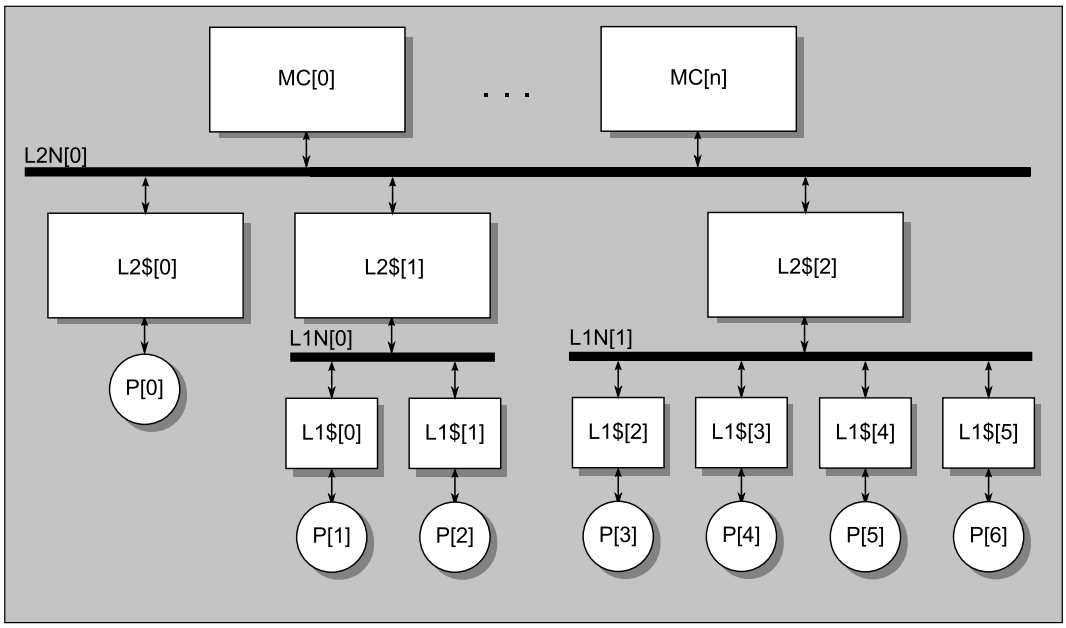


Figure 6.3: *An example of an unbalanced multi-cluster configuration.*

Each cache-distance identifier is a bit-mask which is composed of the encoded locality of each context, defined by the algorithm listed in Algorithm 1. The bit mask is stored in a context specific register, discussed in Section 3.1.1, and can be accessed by the L1 cache logic as well as by privileged software. To illustrate how the cache-distance identifiers are allocated consider the unbalanced architecture shown in Figure 6.4.

Starting with the top level of sharing, level 2, the number of connected components, in the example comprising three L2 caches, is counted in order to derive the number of bits required to represent all components at the same level. For each context the level 2 bitmap is the binary representation of the level 2 component which is above it in the hierarchy, using left to right indexing. In the example all contexts connected to processor `P[0]` have `00` as the level 2 bitmap, contexts connected to `P[1]` and `P[2]` have `01` as the level 2 bitmap. This trivial process is repeated at each level of sharing. It should be noted that all cache-distance identifiers must be of the same length, even when the hierarchy is unbalanced, as illustrated by `P[0]` in Figure 6.4.

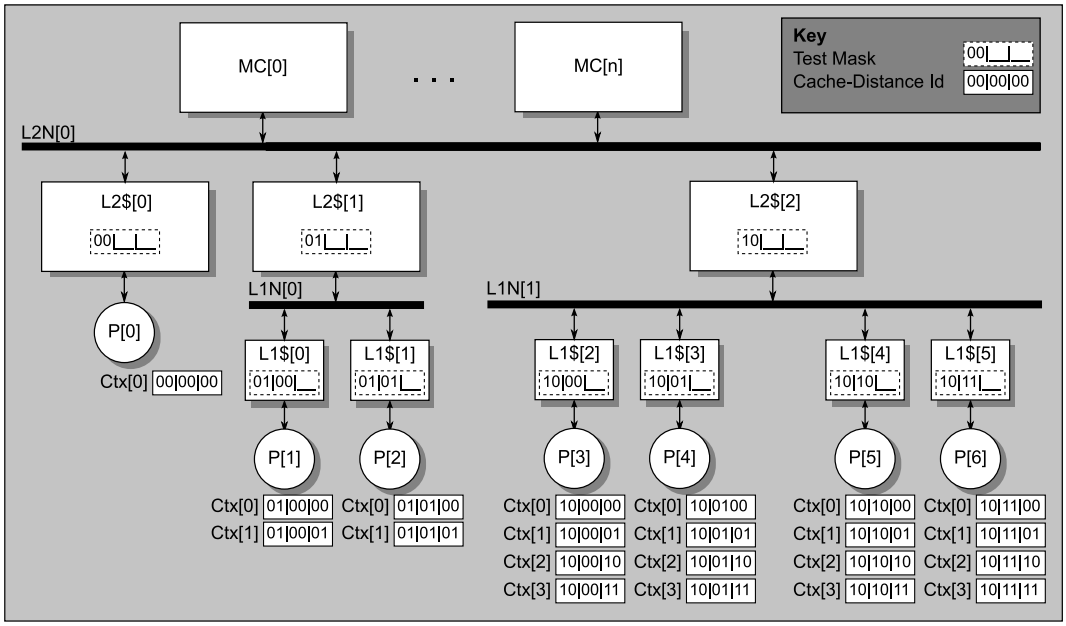


Figure 6.4: *The allocation of cache-distance identifiers to contexts in an unbalanced multi-cluster architecture.*

6.2.5 Hardware Support for Locality

In order to support lightweight threading within the CMC architecture it must be possible to locate idle contexts across the whole chip, and then be able to distribute tasks to available idle contexts.

Locating Idle Contexts

Locating idle contexts within the CMC architecture is done using the same ring structure found in the single bus architecture. Each core on the chip is connected to two neighbouring cores creating a single ring network. When contexts become idle the cache-distance identifier is placed onto the ring or, if no space exists on the ring, into a local token pool. Idle contexts, as previously mentioned in Section 3.1.3, are located by polling the ring network for tokens using the `TRQ` instruction.

A single ring network has two major disadvantages when connecting all of the processing cores in a CMC architecture. Firstly the latency for a cache-distance token to complete a rotation of the ring is equal to the number of cores connected to the ring. With the possibility of integrating hundreds of cores this latency can

be significant, however, the actual latency is dependant on the requirements of the token request instruction. The second disadvantage is that the ring is not fault tolerant. Any damage to the structure during manufacturing will remove the ability to locate idle contexts using the ring. The ring does however provide a simple mechanism that allows multiple cores to poll for and release idle contexts concurrently and is retained for this reason within the CMC architecture.

Distributing Tasks

To support task distribution within the CMC architecture each shared cache must be able to forward task-setup data either up or down the hierarchy so that an idle context, even in a different cluster, can receive it. Each level of shared cache in the architecture contains logic to make a simple comparison of the cache-distance identifier within the task-setup data, and that stored in the caches test mask. The task-setup data is forwarded up the hierarchy until the test mask matches the cache-distance identifier, the task is then moved down the hierarchy of caches based on the values in each successive bitmap within the cache-distance identifier.

Suppose core `P[1]`, in Figure 6.4, upon executing a `TRQ` instruction receives the cache-distance identifier `[101101]`. The forking code executing on the core packages the task-setup data into registers `o0 - o7`, and executes a `THJ` instruction. The L1 cache logic checks that the cache-distance identifier is not within its local group of contexts, using the test mask `[0100--]`, and, in a similar manner to load/store instructions missing in the L1 cache, arbitrates for access to the bus, `L1N[0]`. The task data, consisting of 8 32-bit registers, matches the size of a cache line, and so task distribution reuses the logic already required for cache coherence.

When the interconnect network is a bus, a `THJ` transaction can be snooped by all the caches and consumed by a cache where the test mask matches the cache-distance identifier. When the interconnect network is a crossbar, the `THJ` transaction must be forwarded to the higher level shared cache, where logic then determines if the `THJ` should be forwarded up or down the hierarchy. In this example the L2 cache, `L2$[1]` buffers the `THJ` transaction and the task-setup data, as the cache-distance identifier `[101101]` does not match the test mask `[01----]` and so the transaction must be passed onto the next level bus, `L2N[0]`.

When the THJ transaction is placed onto the top level bus, L2N[0], the L2 cache, L2$[2], is able to match the cache-distance identifier [101101] with the test mask [10----]. The THJ transaction is subsequently forwarded down the hierarchy, using the same process, until it arrives on bus L1N[1] and is consumed by the L1 cache L1$[5]. The cache is then able to wake the relevant context in core P[6] which becomes runnable, and will begin processing the distrubuted task when the context is next scheduled.

The delay associated with shipping a task across the chip is related to the distance that the task is being shipped. A longer delay will be associated with shipping a task to a core in a remote cluster when compared to shipping a task within the same cluster. This delay is acceptable however as distributing the task to a remote core is reserved for tasks that exhibit poor locality or that are sufficiently independent to achieve benefits from running in separate caches.

6.2.6 Software Support for Locality

The lightweight task distribution mechanism is exposed to software via the TRQ instruction so that parallel applications can be optimised to best utilise the CMC architecture.

Token Request Semantics

The TRQ instruction has been modified to allow programs to express a preference for how near or far away in the cache hierarchy a shipped task should be distributed. The TRQ instruction is of the *register* form, see Appendix A.1.1, which is composed of two input operands Ra and Rb and a result register Rc. The first operand is used to define the number of cycles that the TRQ operation is allowed to poll for a token. The TRQ operation returns either a token, containing the cache-distance identifier, or -1 in the result register depending on whether a suitable token is found or not. The second operand is used to define the preferences for selection of a token, and is composed as shown in Figure 6.5.

The operand can be interpreted as either a *mandatory* or *preferential* set of arguments. The TRQ instruction will try to satisfy the arguments of the operand during the poll cycles supplied within the first operand. If no token of any sort is

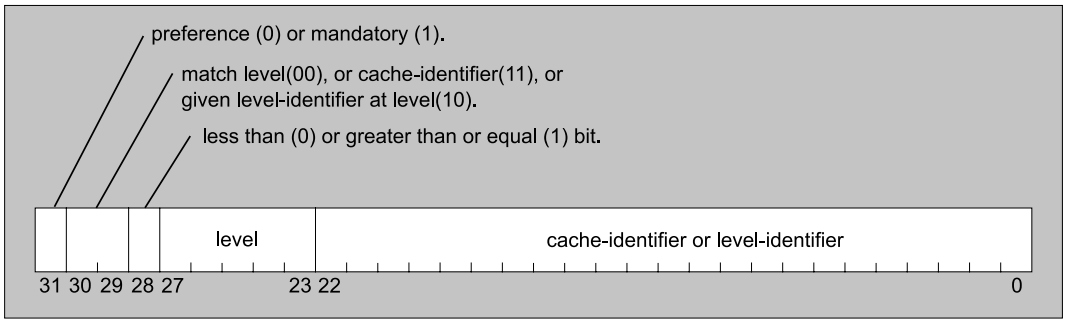


Figure 6.5: *The `TRQ` semantics allow preferences for token selection which are exposed within the preference operand (`Rb`)*

available it will return -1. As soon as a token matching the preferences is found that token is returned, curtailing the polling period. If other tokens are found during the polling period and the arguments are supplied as *preferential*, then each token is held until another is found, at which point the newly found token is held and the previously held token is released back onto the token ring. If no matching tokens are found when the polling period expires the held token is returned.

The semantics allow software to select tokens that are related to a given cache-distance identifier, that are a given level distance from the executing context's cache-distance, or that are in a particular cluster at a given level. Because the `TRQ` instruction stores the resulting cache-distance identifier in register `Rc`, the software can use this value in future distribution operations. However, the software can not generate a *cache-distance* token to distribute a task to, the returned identifier is stored in a privileged internal register which is used by the following `THJ` or `THB` instruction. Additional instructions within the instruction set allow software to enquire about the number of levels of sharing and the number of components at each level, relative to the context executing the instruction. These values are hardwired into control registers within each processor.

To utilise the token semantics two initial distribution methods were developed; cluster affinity and remote-local distribution. Using the `TRQ` semantics it is also possible that other scheduling schemes described in the literature could be implemented within software, in particular balance-set scheduling [43] and sampling-based and electron-based policies [164] and go some way to approaching quality-of-service schemes [73].

Remote-Local Distribution

Remote-local distribution is a simple policy that allows a thread distributing work to decide whether the task should be forked to a local context, within a cluster at a given sharing level to improve data locality, or to a remote context in a remote cluster at a given sharing level to improve load-balancing. This scheme is used by software to keep threads either local, for example when dividing work on a shared array of data, or to ship threads away from the distributing thread to avoid unnecessary cache interference that may delay the progress of the distributing thread.

A disadvantage to the remote-local distribution policy is that remote tasks are sent to arbitrary remote clusters, based on the order they are acquired using the token-request `TRQ` instruction. This can lead to work imbalance, where a number of remote tasks are distributed to the same cluster and potentially clusters remain idle. The remote-local scheduling is best used when a large number of worker threads need to be distributed by a single distributing thread. The distributing thread can therefore opt to distribute tasks to remote clusters or cores to allow itself to progress without being impeded by time-sharing a core's pipeline.

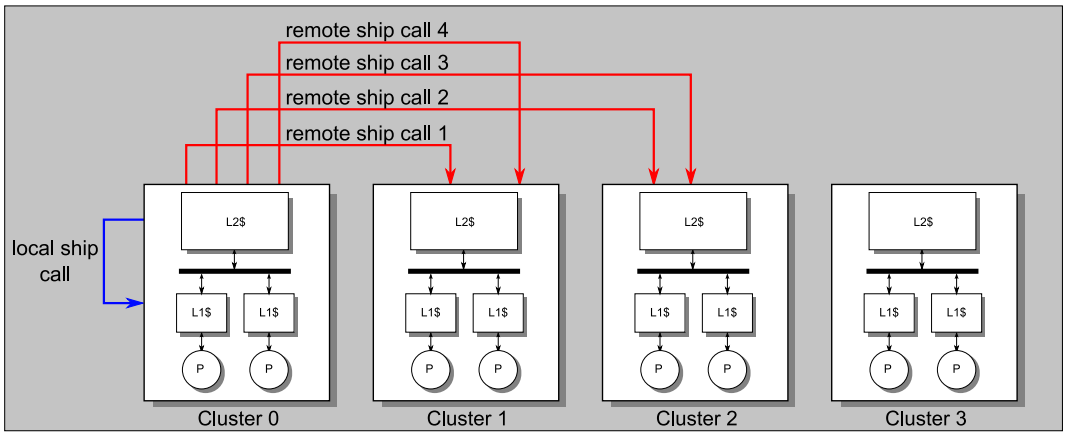


Figure 6.6: *Remote-local distribution allows a program to fork a task to a context within either a local or a remote cluster. Even though four remote threads are forked there is no guarantee all clusters will receive work.*

Cluster Affinity

Cluster affinity distribution allows a program to ship tasks to a context within a specified cluster of processors at a given sharing level. The policy enables software to distribute tasks to all clusters at a given sharing level, ensuring that all clusters are utilised within an architecture leading to a better load-balance, see Figure 6.7. Additionally this is beneficial when running multiple independent application threads. By running each application in isolation, each restricted to a separate cluster, cache interference can be avoided within the smaller, lower level, shared caches whenever possible.

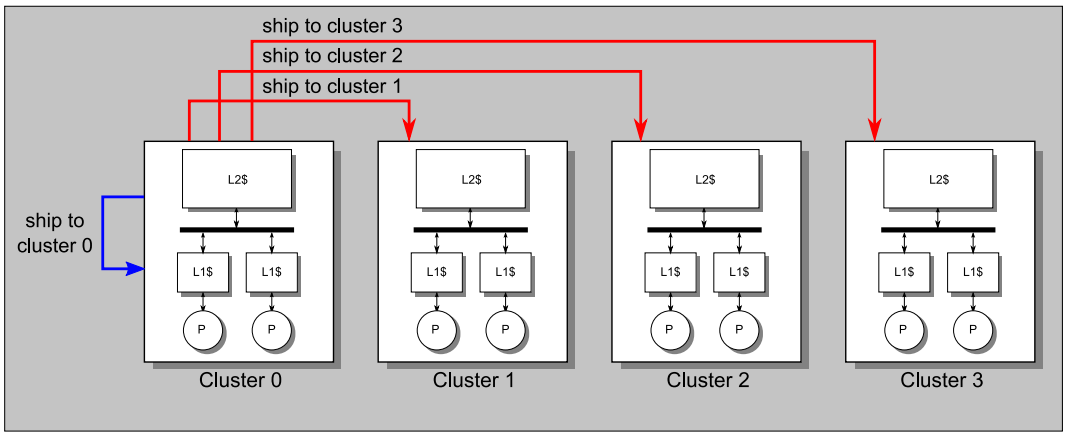


Figure 6.7: *Cluster affinity allows a task to be distributed to a specified cluster at a given sharing level.*

Software can also use cluster affinity as a means of keeping a thread within the same cluster throughout its lifecycle. On a heavily loaded system, many threads will be competing for a limited number of hardware contexts to execute on. A global thread scheduler is responsible for ensuring all threads make forward progress, and when required will force threads to yield. When those threads are subsequently rescheduled cluster affinity can be used to ensure that they are rescheduled within the same cluster in order to benefit from previously cached data.

6.3 Summary

This chapter has described how locality can be exploited within a CMC architecture. In particular an extension to the `TRQ` instruction was presented that allows software to exploit locality by controlling the affinity of distributed tasks. Two simple examples of the use of the extended `TRQ` instruction for locality based task distribution were presented.

The next chapter presents and analyses results from experimentation using the coherence protocol introduced in Chapter 4, the hardware support introduced in Chapter 5, and finally locality aware task distribution introduced in this chapter.

CHAPTER 7

Results and Analysis

Previous chapters have introduced a cache coherence protocol for multiple levels of shared cache, a CMC architecture built to support this protocol and locality based task distribution to take advantage of cache locality within the architecture. This chapter analyses the architecture, protocol and distribution mechanism by exercising a cycle-level simulated system using parallel benchmarks.

7.1 Experimental Method

Accurately evaluating the performance of the CMC architecture, the coherence protocol and the locality distribution mechanism requires a way of simulating the system and exercising its components with workloads likely to reflect those used in real parallel systems. In this section the simulation environment and the benchmark applications used in this study are described.

7.1.1 Simulation Environment

To evaluate both the protocol and the architecture built to support it, the `jamsim` framework was extended with additional components. These components include a modified cache component, able to express the seven states of the PIMMS protocol, see Table 4.2, and both a bus and crossbar based interconnect able to implement the protocol transitions.

For each simulation the architecture is configured with, unless otherwise stated, the parameters listed in Table 7.1. The processor, the interconnect and the memory hierarchy are simulated using cycle-level models to account for, and allow further analysis of, the many interactions, stalls, queue delays and blocking associated with the architecture. Additionally the simulation platform has been extensively instrumented to extract statistical data from each of the studied components and overall performance metrics.

Component	Parameters
L1 caches	16KB, 4-way set-associative, access 1 cycle, 4 entry core- & memory-bound queues.
L2 cache	2MB, 8-way set-associative, access 8 cycles, 4 entry core- & memory-bound queues.
L3 cache	4MB, 16-way set-associative, access 32 cycles 4 entry core- & memory-bound queues.
Off-chip Memory	2GB, access 100 cycles
L1-L2 bus	8 phase, memory led split-transaction protocol, L2 clock
L2-L3 bus	8 phase, memory led split-transaction protocol, L3 clock

Table 7.1: *Configuration of the simulated cache hierarchy.*

The Java benchmark applications selected are executed within the ported version of the Jikes RVM which is hosted natively on the simulator. The use of the JaVM allows Java threads to be mapped onto the underlying hardware thread distribution mechanism. The benchmark applications are statically compiled into the Jikes RVM bootimage using the highest level of optimisation (`-O2`). This avoids the cost of dynamic compilation and optimisation of the benchmark classes during execution and as a consequence reduces the impact of noise and interruptions by JaVM threads to the application threads at runtime, allowing a more intuitive reasoning about the performance.

7.2 Benchmark Descriptions

In order to exercise the architecture and stress the coherence protocol a set of multi-threaded applications were selected from two benchmark suites: Doug Lea's Fork/Join package [93] and the JavaGrande Forum benchmark suite [144]. The parameters used for each benchmark have been chosen to avoid the side-effects of garbage collection during execution, again avoiding unnecessary JaVM activity.

7.2.1 Fork/Join Benchmarks

The fork/join benchmarks have been selected from a set of nine *demonstration* applications used to study parallel application performance using a Java work-stealing framework. Out of the nine benchmark applications three were discarded, *Microscope* because the code was heavily interleaved with a graphical user interface, *Heat* because standard parameters consumed too large a simulation time, and *NQueens* because results and timings are non-deterministic, due to the nature of its multiple solution strategy.

The fork/join benchmarks are supplied at runtime with the number of threads available to process tasks; for each simulation configuration this is set to the number of hardware supported contexts. The following sections briefly introduce the six benchmarks selected.

Fibonacci

The `fibonacci` benchmark calculates the n^{th} fibonacci number by recursive parallel decomposition. The initial number is decomposed into two parallel tasks to calculate the $(n-1)$ and $(n-2)$ numbers. This decomposition is done recursively until the value of n falls below a threshold, at which point it is calculated sequentially. The results of the decomposed values are then successively combined to form the total result. Each task of the decomposed pair, $(n-1)$ and $(n-2)$, must wait for the other to complete before combining the results and completing the n task that they themselves were recursively divided from.

MatrixMult

The `matrixMult` benchmark performs a parallel divide-and-conquer matrix multiplication. The matrices A and B are divided into quadrants and then multiplied using Equation 7.1.

$$\begin{pmatrix} A_{1,1} & A_{1,2} \\ A_{2,1} & A_{2,2} \end{pmatrix} \times \begin{pmatrix} B_{1,1} & B_{1,2} \\ B_{2,1} & B_{2,2} \end{pmatrix} = \begin{pmatrix} (A_{1,1} \times B_{1,1}) & (A_{1,1} \times B_{1,2}) \\ (A_{2,1} \times B_{1,1}) & (A_{2,1} \times B_{2,1}) \end{pmatrix} + \begin{pmatrix} (A_{1,2} \times B_{2,1}) & (A_{1,2} \times B_{2,2}) \\ (A_{2,2} \times B_{2,1}) & (A_{2,2} \times B_{2,2}) \end{pmatrix} \quad (7.1)$$

The matrices on the right hand side are recursively divided into smaller quadrants until reaching the threshold set in the benchmark, at which point the leaf multiplications are calculated using sequential code. Synchronisation is required to recombine all the quadrant results in order to produce the result matrix.

Jacobi

The `jacobi` benchmark performs iterative relaxation on a matrix mesh. The initial mesh is configured with the value 1 in all edge elements and 0 in all other elements. The complete mesh is represented internally as a tree structure and relaxation is carried out on each of the leaf nodes within the tree. Each leaf node contains a subsection of the mesh upon which nearest neighbour averaging is carried out sequentially. Leaf nodes are processed in parallel tasks, with synchronisation occurring where edge elements overlap two leaf nodes. A defined number of nearest neighbour averaging iterations is performed over the total mesh, until the number of iterations expires or the result converges.

LU

The `lu` benchmark performs a matrix decomposition of a randomly filled matrix, into the product of two triangular matrices, *Lower* and *Upper*, and is a Java version of the well known Linpack benchmark. The actual composition of the algorithm is beyond the scope of this study, suffice to say that the decomposition is calculated again in a divide and conquer manner. Eventually the division creates sub-matrices whose granularity falls under a threshold, at which point

the LU decomposition is calculated sequentially inside that matrix, and multiple calculations are performed in parallel.

Integrate

The `integrate` benchmark computes integrals using a recursive Gaussian quadrature. Essentially this calculates the area under a curve using a finite approximation, by dividing the area into sections and calculating rectangular areas. The function of the curve, for which the integral is being calculated is listed in Equation 7.2.

$$(2i - 1)x^{(2i-1)} \tag{7.2}$$

MergeSort

The `mergeSort` benchmark performs a parallel merge/quick-sort of a set of integers. The complete array of integers is recursively subdivided into smaller sets of integers, until the set size falls below a threshold at which point the standard quick-sort algorithm is applied. Quick-sort of multiple sets occur in parallel, and the results are then merged into larger sets until the whole array has been sorted.

7.2.2 Multithreaded JavaGrande Benchmarks

The JavaGrande multithreaded benchmark suite consists of three sets of benchmarks; low-level, kernel, and application codes designed to evaluate parallel approaches to standard computationally intensive problems. From these three sets, one low-level and three kernel tests were selected. The benchmarks have largely been re-programmed with reference to the sequential JavaGrande benchmarks, which in turn were re-coded versions of the Splash-2 benchmark suite [169]. Parallel work is distributed statically to the number of threads passed to the benchmarks as a parameter. Unlike the fork/join benchmarks, no work stealing occurs, and so each thread completes the whole portion of the work given to it.

BarrierBench

The `barrierBench` benchmark is a low-level benchmark designed to measure the performance of barrier synchronisation. Internally the benchmark creates a number of threads which loop for a given number of iterations and attempt to synchronise on two types of barrier, a simple shared counter and a lock-free tournament barrier.

Series

The `series` benchmark calculates the first n Fourier coefficients of the function listed in Equation 7.3, over the interval $[0, 2]$. The benchmark consists of a large loop over the Fourier coefficients, however, each iteration of the loop is independent of every other iteration and the work is simply divided and distributed to the parallel threads.

$$f(x) = (x + 1)^x \tag{7.3}$$

SOR

The `sor` benchmark performs 100 iterations of successive over relaxation on a $N \times N$ grid. The benchmark contains three loops, the outer iteration loop and two inner loops over the row elements to process the relaxation. In order to parallelise the algorithm the elements in the grid are processed using a "red-black" ordering scheme, which allows the inner loops to be partitioned between the threads. Synchronisation occurs at the end of each iteration to ensure that all red (black) elements have been updated before attempting to update the black (red) elements.

Crypt

The `crypt` benchmark performs IDEA (International Data Encryption Algorithm) encryption and decryption on an array of N bytes. The benchmark contains two principle loops whose iterations are independent and can therefore be

partitioned between the threads in a block fashion.

7.2.3 Benchmark Parameters

For completeness the parameters used for each benchmark, unless otherwise stated alongside results, are listed in Table 7.2. Each benchmark was run using N threads, where N equals the number of hardware contexts supported by the architecture being simulated.

Benchmark	Type	Parameters
`barrierBench`	low-level	size = 0
`fibonacci`	kernel	number = 39, threshold = 20
`matrixMultiply`	kernel	matrix = 1024×1024, granularity = 128
`jacobi`	kernel	matrix = 1024×1024, steps = 128
`lu`	kernel	matrix = 512×512
`integrate`	kernel	low = 1, high = 42, exponential = 5, tolerance = 0.001
`mergeSort`	kernel	sort array = 5,000,000 integers
`series`	kernel	size = 100
`sor`	kernel	matrix = 256×256
`crypt`	kernel	500,000 bytes

Table 7.2: *Benchmark parameters used during experimentation.*

7.3 PIMMS Coherence Protocol

Having outlined the simulation configurations and the benchmarks used to exercise the architecture and the coherence protocol, this section looks at various aspects of the protocol, working under simulation, to assess its correct function.

7.3.1 Coherence Transactions

As mentioned in Section 4.3.2, nine coherence transactions coordinate all of the on-chip shared memory coherence. Coherence messages propagate both up and down the multi-level cache hierarchy to ensure that the cached representation of memory seen by all contexts is consistent.

Coherence transactions are tracked during simulation when they appear in the interconnect. This tracking provides an insight into both the composition of the coherence traffic and the peak utilisation of the interconnect. Figure 7.1, shows this utilisation, during execution of the `lu` benchmark. As the system is configured as a two cluster CMC, there are two bus networks, `L1N[0]` and `L1N[1]`, connecting the L1 private caches of the 64 cores within each cluster and a higher level bus network, `L2N`, connecting the two L2 caches.

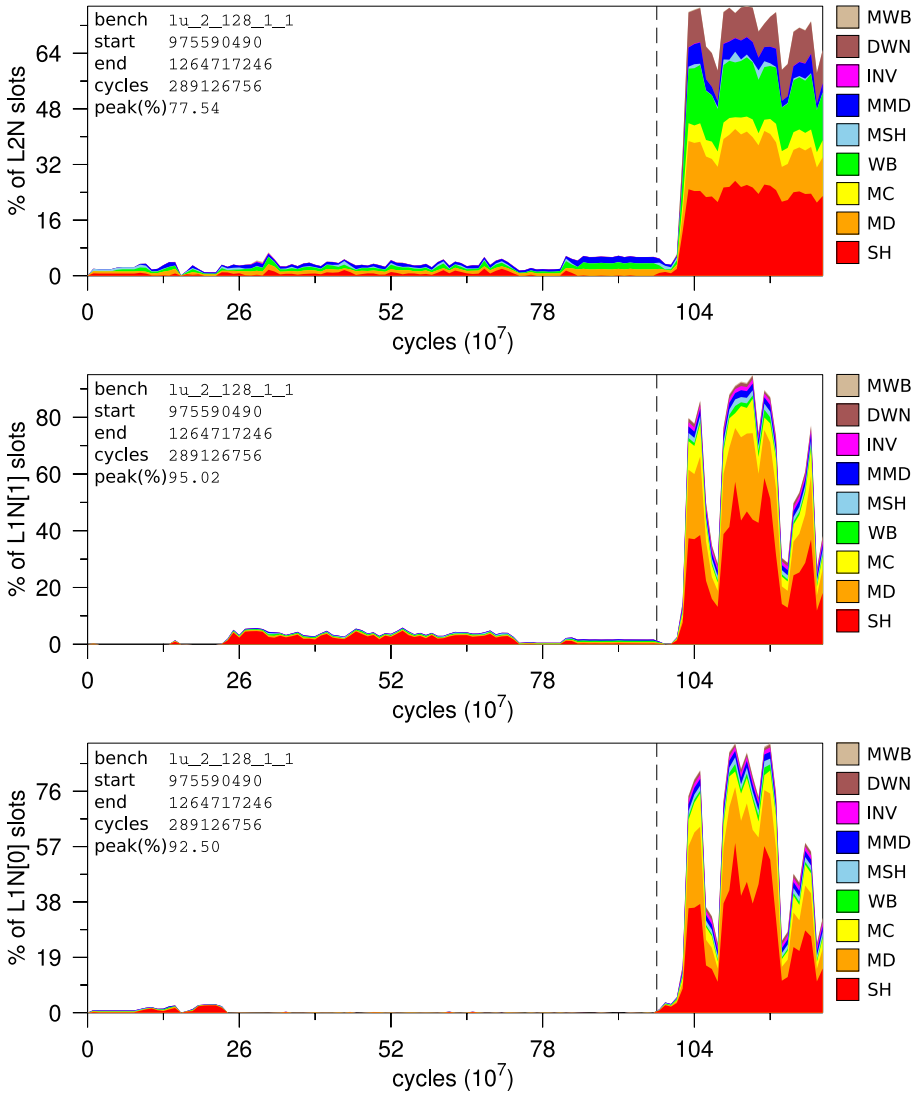


Figure 7.1: *Bus utilisation during execution of the* `lu` *benchmark. The architecture is configured as a symmetric 2 cluster × 64 processors × 1 context CMP.*

The area under the utilisation curve is divided, by percentage, into the constituent coherence message types. SH, MD, MC and WB are memory-bound transactions,

MSH, MMD, INV, DWN and MWB are core-bound, and are described fully in Section 4.3.2.

As the benchmarks are Java classes, the period during which the JaVM is booted and all supporting classes are loaded is significant, in Figure 7.1 this period lasts until cycle 975,590,490. After this point the benchmark execution begins and the coherence traffic increases significantly. The benchmark executes on all of the processors, and the bus utilisation increases to peak at 95.02% on the `L1N[1]` bus.

The coherence traffic on buses `L1N[0]` and `L1N[1]` mainly consists of transactions satisfied either by a load from the L2 cache or a transfer of ownership/permissions from another L1 cache. This is noticeable as the number of the SH, MD and MC request transactions far exceeds the corresponding memory oriented responses.

7.3.2 Four-phase Transactions

As mentioned in Section 4.3.4, four-phase transactions are necessary in order to maintain coherence across the cache hierarchies in a CMC architecture. The number of four-phase transactions can be calculated from the combination of MWB and DWN transactions, which are only triggered during the *action* phase of a four-phase transaction.

In Figure 7.1 the number of four-phase transactions peaks at around 10% of the total transactions seen on the level 2 bus, some 130,711 MWB and DWN transactions during a period of 10 million cycles, in which 1.25 million level 2 bus slots are available.

The `lu` benchmark, the execution of which generated Figure 7.1, performs synchronisation as it combines results of sub-matrix decomposition to form the whole. During this process `synchronized` methods inside the benchmark ensure that locks are gained on the results prior to combining them, so some four-phase transactions are generated. Figure 7.2, however, better illustrates the four-phase transactions, this time generated during execution of the `barrierBench` benchmark. As the benchmark is simply attempting to synchronise on barriers a larger portion of the coherence traffic visible on all buses is related to four-phase transactions.

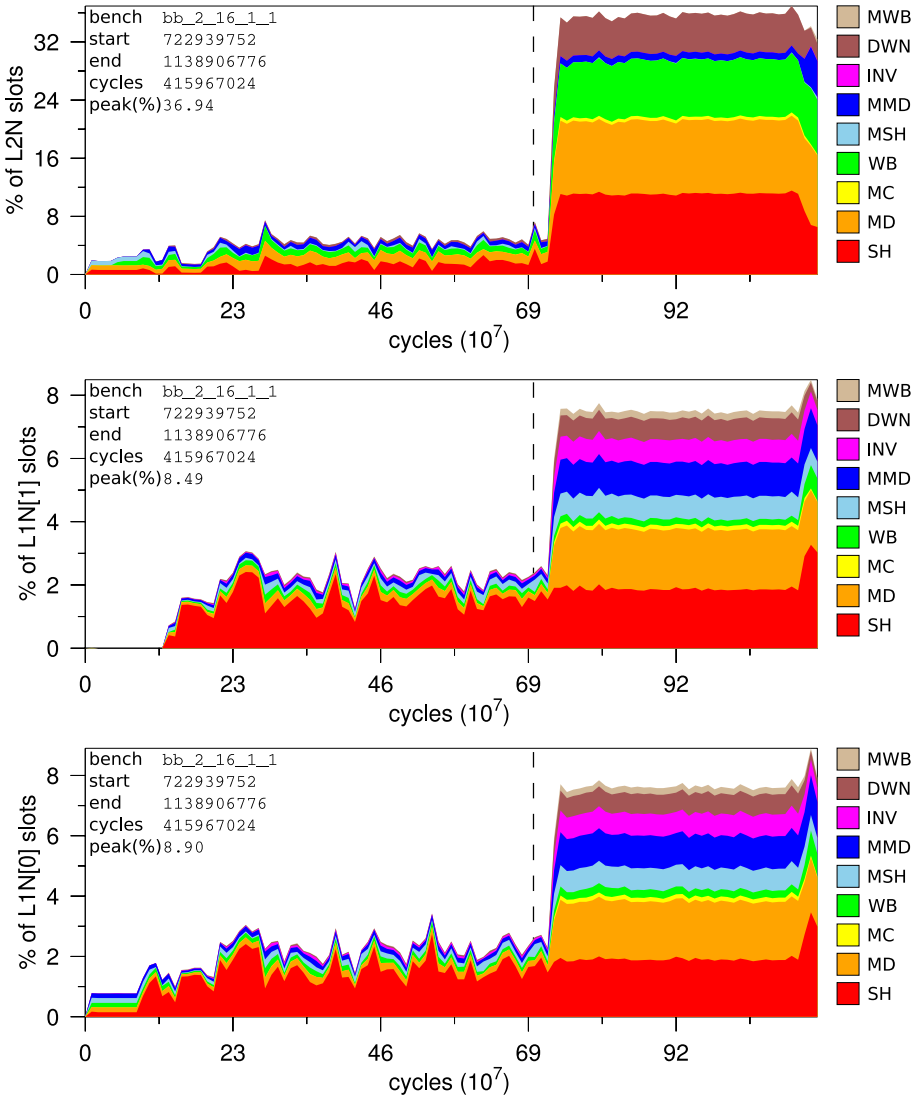


Figure 7.2: *Coherence traffic generated during the JavaGrande* `barrierBench` *benchmark, on a symmetric 2 cluster × 8 processors CMP.*

7.3.3 Interconnect Latency

Another factor which is introduced with the extension to multiple clusters is the effect that memory sharing across clusters has on the latency of the interconnect. The graph, Figure 7.3, shows the average network latency during discrete periods of 1 million cycles on each of the shared buses in a two cluster architecture during execution of the `lu` benchmark. The latency of each transaction is calculated from the time that the request arbitrates for the bus to the point at which the response is received.

Prior to execution of the benchmark, denoted by the dashed line, the average latency on all of the buses is fairly erratic as the architecture is booting the JaVM and loading and compiling all necessary classes, requiring frequent calls to memory. Immediately before the benchmark is invoked the latency increases significantly for a period of around 160 million cycles. This increase in latency, upto 240 cycles and 160 cycles on the level 2 and level 1 buses respectively is caused by memory allocation as the benchmark begins to create data structures. Once the benchmark is executing however, the average latency decreases to around 10 cycles on the level 1 buses and 40 cycles on the level 2 bus. These latencies are almost as low as is feasible, given that the minimum time for a single transaction to complete is 8 cycles on the level1 bus and 32 on the level 2 bus.

The observed latency is related to the distance that each transaction is required to travel in the hierarchy in order to gather the requested data. In this example, most requests are being satisfied directly from the shared cache without having to traverse the inter-cluster bus.

7.3.4 Negative Acknowledgment

During periods of heavy coherence traffic on the interconnects, transactions may not be able to commit, which can occur when transaction queues are at their capacity. A negative acknowledgment signal is sent to the cache attempting the transaction, and the cache must subsequently re-attempt the transaction.

Figure 7.4 shows the percentage of the total bus slots negatively acknowledged over the execution of the `fibonacci` benchmark on a 16 core CMP.

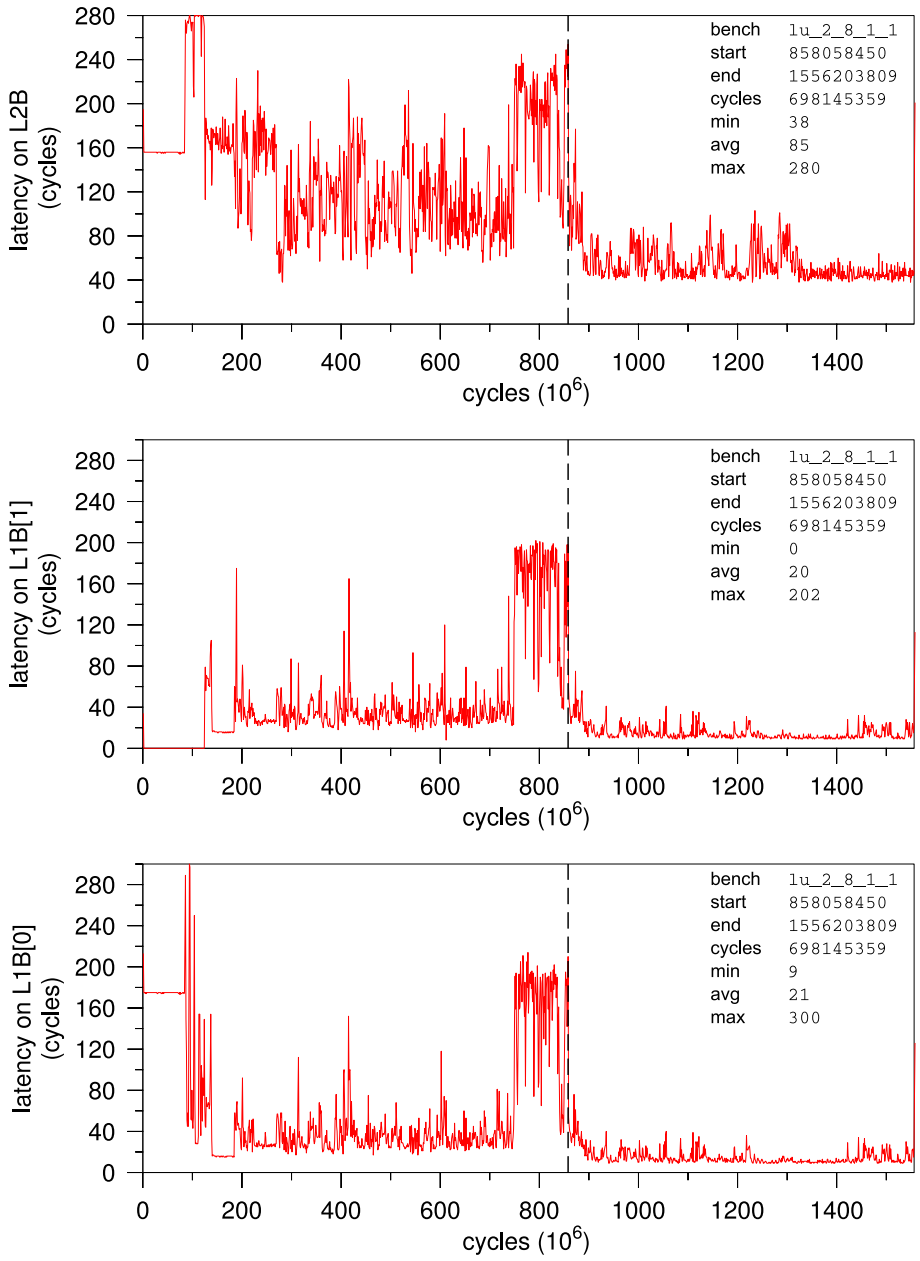


Figure 7.3: *Average latency of bus transactions during execution of the lu benchmark on a 2 cluster × 4 processors CMP.*

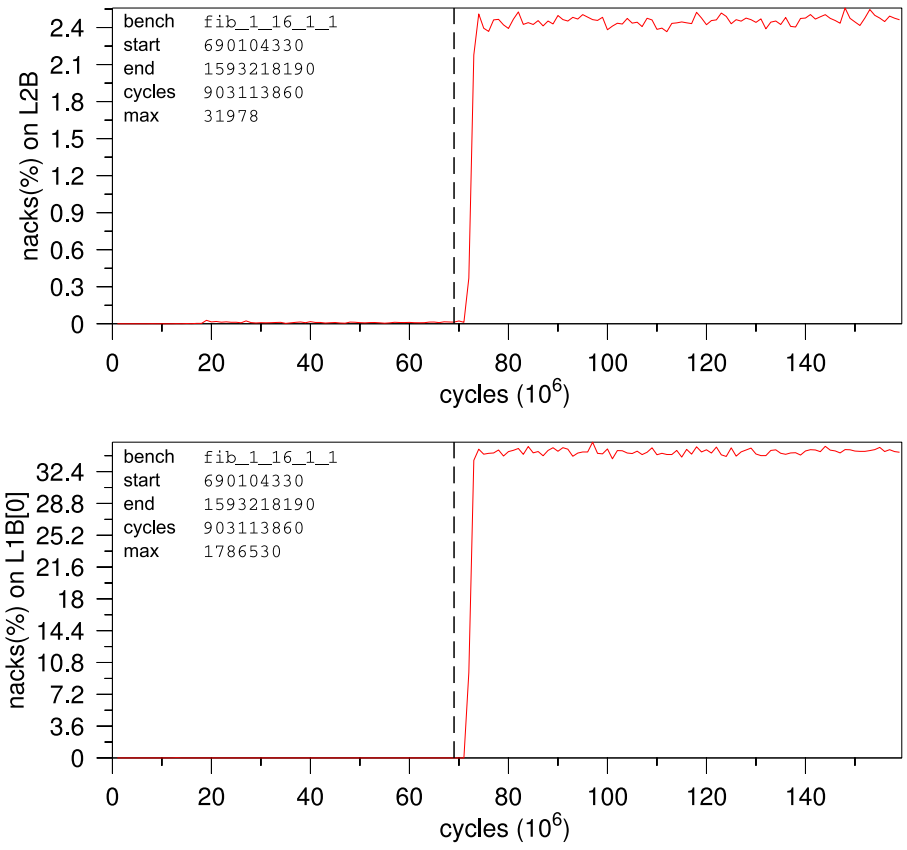


Figure 7.4: *Negative acknowledgments blocking a transactions progress as a percent of available bus slots, for a single cluster 16 processor CMP running the* `fibonacci` *benchmark.*

7.3.5 Non-Sinkable Queue Rotation

As introduced in Section 5.4.4, each cache's non-sinkable transaction queue is split into a passive and active sub-queue. This allows requests in the non-sinkable queue to be re-ordered when a transaction at the head of the queue is blocked due to address locking or resource contention.

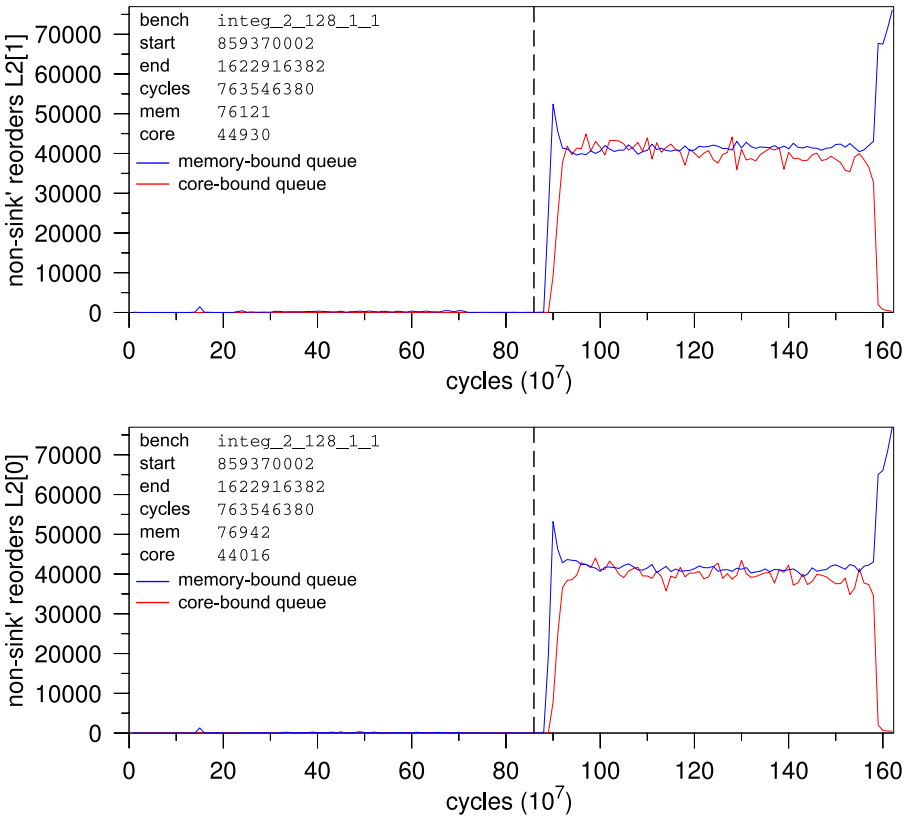


Figure 7.5: *Non-sinkable passive/active queue reordering (per 10 million cycles) for both L2 caches, in a 2 cluster × 64 processors CMP, executing the* `integrate` *benchmark.*

Figure 7.5, shows the number of reordering events between the sinkable and non-sinkable queues, during the execution of the `integrate` benchmark on a single cluster CMP with 128 cores. These events are counted when a transaction successfully commits that was behind an aborting transaction that has been reordered.

7.3.6 Effect of Inclusion

A limitation in the current implementation of the PIMMS coherence protocol is that each shared level cache in the multi-level hierarchy maintains cache-line inclusion[1]. This introduces redundancy within the shared level caches and reduces their efficiency. Lines in the Valid Shared, Modified Shared, and Modified Stale states are redundant in each shared-level cache.

Figure 7.6 illustrates the state of the lines within each of the shared caches, at 1 million cycle intervals, during execution of the `fibonacci` benchmark. The architecture is configured as a 2 cluster CMP where each cluster contains a 1MB level 2 shared cache, and 8 processing cores connected to 16KB instruction and data caches. A 4MB level 3 cache is shared by both clusters. During the benchmark phase, from around 750 million cycles, when all of the cores are executing code, on average 12.5% of the L2 cache's lines are redundant, and 40-45% of the L3 cache lines are redundant. In the absence of any code or data sharing between contexts the redundancy in the caches is directly proportional to the size of the caches directly below, and can be calculated, using Equation 7.4, where S_l is the size of the larger shared cache, S_s is the size of the smaller caches, and R is the percentage of redundant lines in the larger cache due to inclusion.

$$R = 100 \times \frac{\Sigma\left(S_s\right)}{S_l} \tag{7.4}$$

Using this calculation the expected redundancy, in the absence of sharing is 25% in the L2 caches and 50% in the L3 cache. The discrepancy between expected and actual redundancy is primarily due to code sharing in the `fibonacci` benchmark and the small amount of data that is shared between the threads.

Looking at inclusion across all of the kernel benchmarks, for a 2-cluster configuration containing 4, 8, 16 and 32 cores per cluster, shown in Figure 7.7, the maximum percentage of redundant lines in the L2 caches remains below 10% when shared by four cores, 20% when shared eight cores, 30% when shared by 16, and below 45% when shared by 32 cores. In most configurations the average

[1] A potential future enhancement to the architecture would be to support tag inclusion. Storing, in a separate cache structure, only the tags of lines that have been modified by a lower-level cache. This is not considered further here.

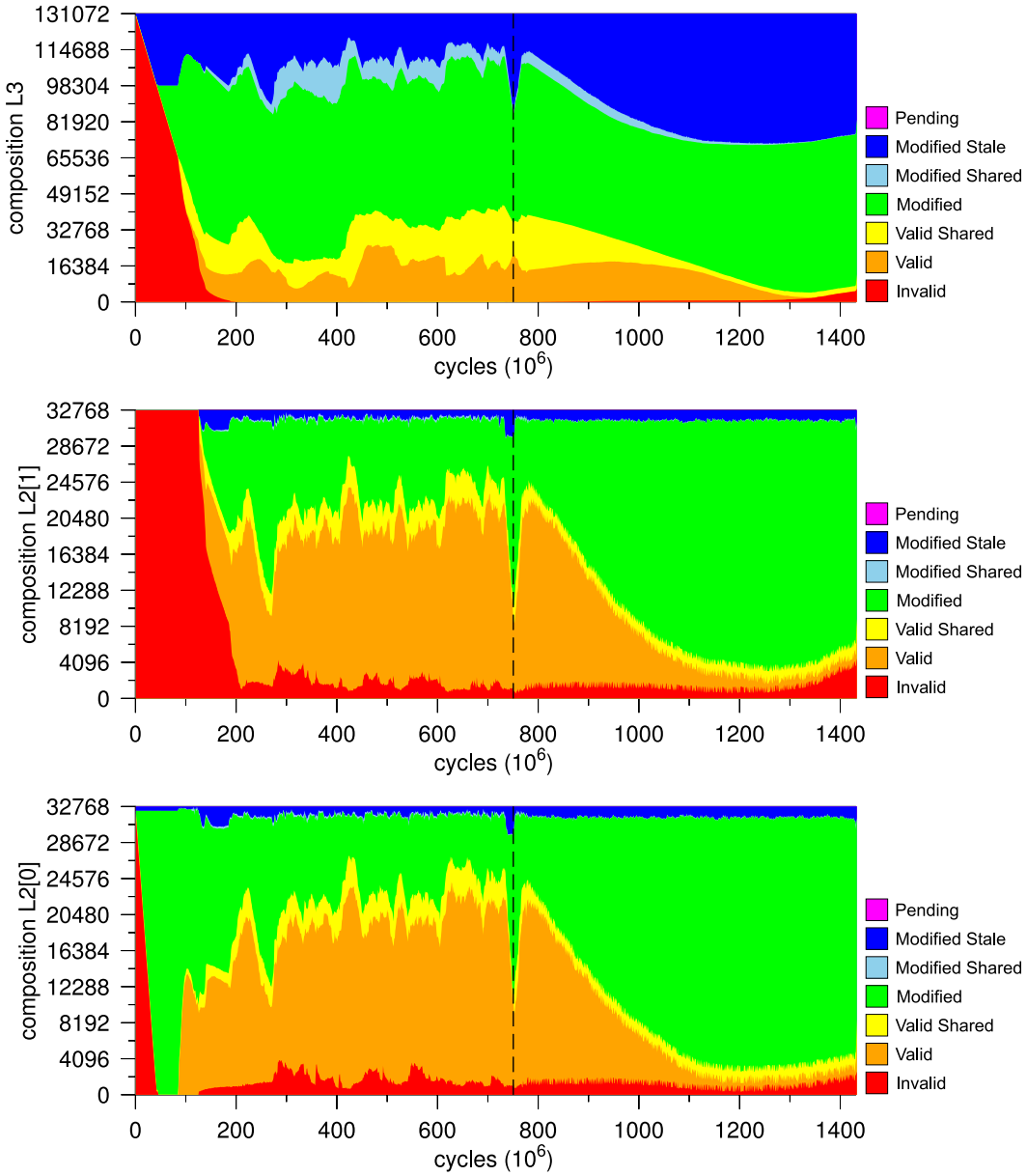


Figure 7.6: *Cache-line state composition during the execution of the* `fibonacci` *benchmark on a 2 cluster × 8 processors CMP.*

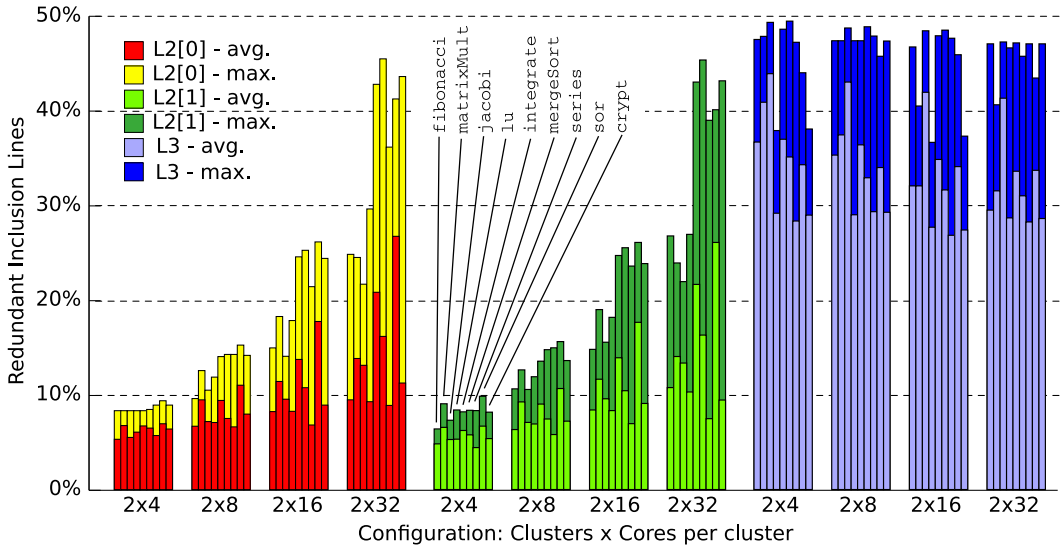


Figure 7.7: *Cost of inclusion measured as the percentage of cache lines containing redundant copies of data, for all nine parallel benchmarks in a 2 cluster CMP.*

redundancy is far less. In the absence of code and data sharing the percentage of redundant lines is calculated as 12.5%, 25%, 50% and 100%.

The percentage of redundant lines in the L3 cache for all configurations, however, peaks far closer to 50%, which is to be expected as essentially a far larger proportion of the L2 cache lines will contain non-shared data.

7.3.7 Protocol Robustness

Finally, whilst no formal analysis has been made as to the correct functioning of the coherence protocol and the architecture under every possible condition, it should be noted that for each configuration of the benchmarks tested, those that did not fail when run using the *perfect* memory simulator, were also able to run using the full cycle-level model of the memory hierarchy. Additionally a significant number of cycles have been executed during the compilation of results, each simulation of each benchmark executes for many billion cycles. During this time the memory hierarchy has remained deadlock free.

7.4 Single Bus CMP Architecture

This section looks at the scaling performance of a single bus CMP. Single bus CMP architectures are attractive to build, primarily because cache coherence across a single bus is well understood, and thus they have, to date, been the subject of most CMP studies. However, as introduced in Section 4.2.1, two main factors limit the scaling of their performance; wire delay and bus contention.

Wire delay is a physical limit and is discussed in detail in Section 1.2.1. Using data presented in the literature [104] it appears clear that connecting many processing cores to a single bus will inevitably require that the bus frequency is reduced to allow the signal to propagate successfully. As outlined in Section 4.2.1 and Table 4.1, even connecting 8 processors to a single bus may require the bus speed to be reduced to $\frac{1}{4}$ of the maximum on-chip frequency. Further reductions will undoubtedly be necessary as the number of cores connected to the bus, and hence its span increases. *Bus contention* also increases as the number of cores added to the bus increases. Contention for the bus, and the subsequent delay caused to a given core's private cache effectively increases the access latency to the level 2 cache. This latency affects access to the memory hierarchy in general, impeding parallel performance gains. Additionally as the number of cores is increased *memory saturation* can occur when the total number of requests generated exceeds the available bandwidth.

To investigate the impact these two factors have on the performance of a single bus CMP architecture the `jamsim` simulation platform was used to combine the simulation of an increasing number of cores and a range of decreasing bus frequencies, shown in Figure 7.8.

Each of the nine parallel *kernel* benchmarks was executed on each configuration of single bus CMP architecture, with the number of threads created by each benchmark equal to the number of cores in the architecture.

7.4.1 Speed-up

The results from these experiments are presented, initially, as speed-up graphs, Figures 7.9 – 7.17. On each graph the vertical axis, representing speed-up, is

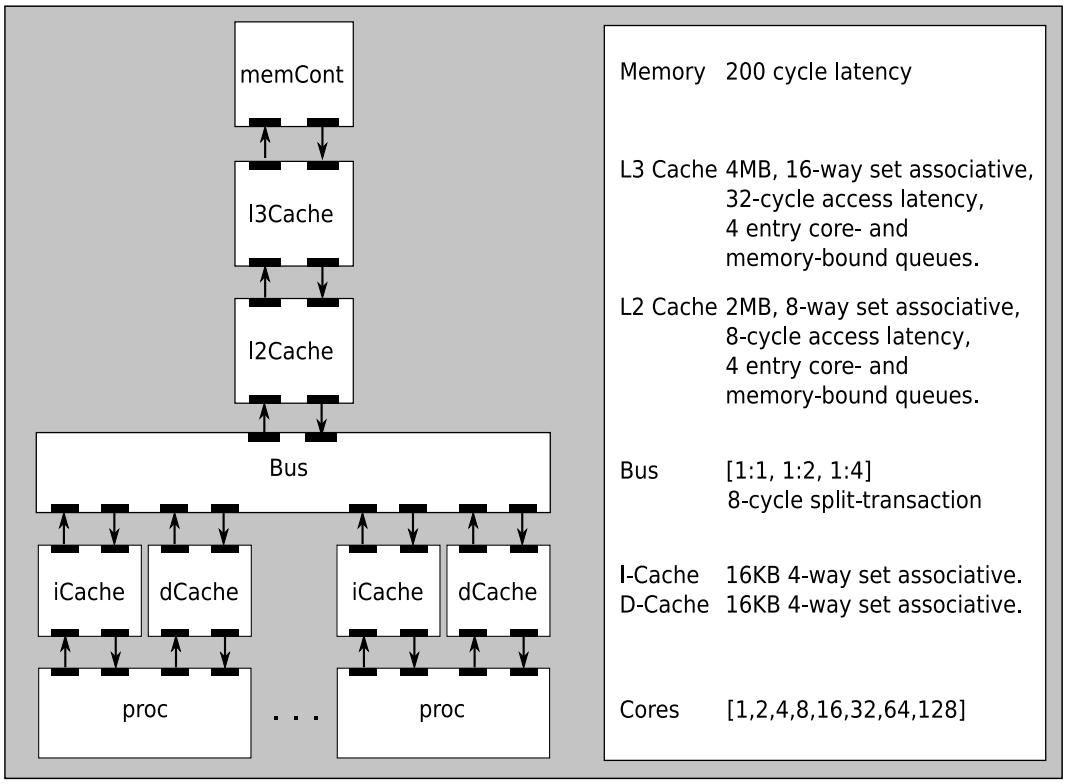


Figure 7.8: *Single bus CMP configuration.*

scaled to the maximum attainable speedup using a simulated configuration with perfect memory[2]. This scale helps to quantify the parallelism inherent in the benchmark, against that which is achieved by the architecture.

$$speedup(n) = \frac{t_1}{t_n} \tag{7.5}$$

Speed-up is calculated using Equation 7.5, where t_1 is the number of cycles taken to execute the benchmark with a single core and t_n is the number of cycles taken with n cores. From the speed-up graphs, the benchmarks can be grouped into three categories. Those that scale well with an increasing number of cores, *near-linear*, those that scale but realise diminishing returns from an increasing number of cores, *diminishing*, and finally those that reach a scaling limit and realise no further returns from an increasing number of cores, *limited*.

In the *near-linear* category are the benchmarks `fibonacci`, Figure 7.9, and `crypt`, Figure 7.17. In the *diminishing* category are the benchmarks `matrixMult`, Figure 7.10, `lu`, Figure 7.12, `series`, Figure 7.15, and `sor`, Figure 7.16. Finally in the *limited* category are the benchmarks `jacobi`, Figure 7.11, `integrate`, Figure 7.13, and `mergeSort`, Figure 7.14.

The following sections look at the impact wire delay, bus contention and memory saturation have on the performance scaling of each of the benchmarks.

[2]The simulator can be configured with no memory hierarchy, i.e. all memory accesses happen instantaneously. Coherence locking is simulated using a simple global lock table.

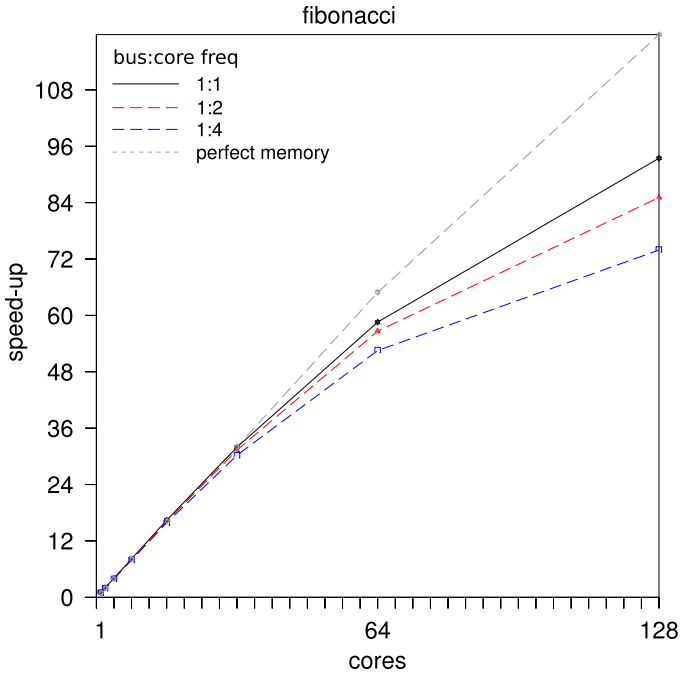


Figure 7.9: *Single bus CMP scaling - `fibonacci`.*

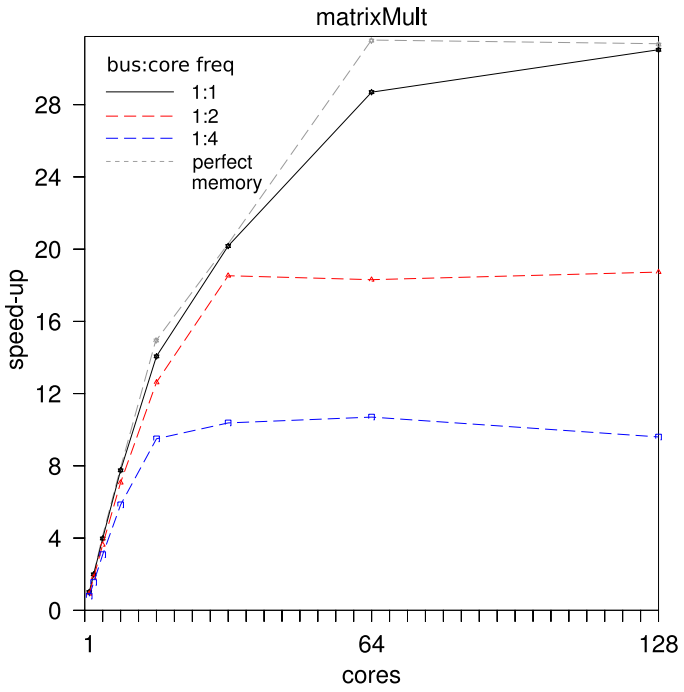


Figure 7.10: *Single bus CMP scaling - `matrixMult`.*

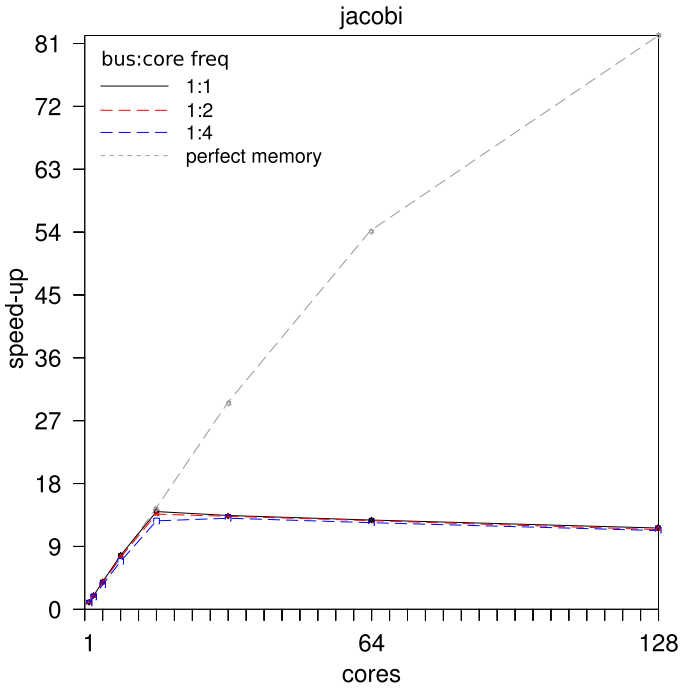


Figure 7.11: *Single bus CMP scaling - `jacobi`.*

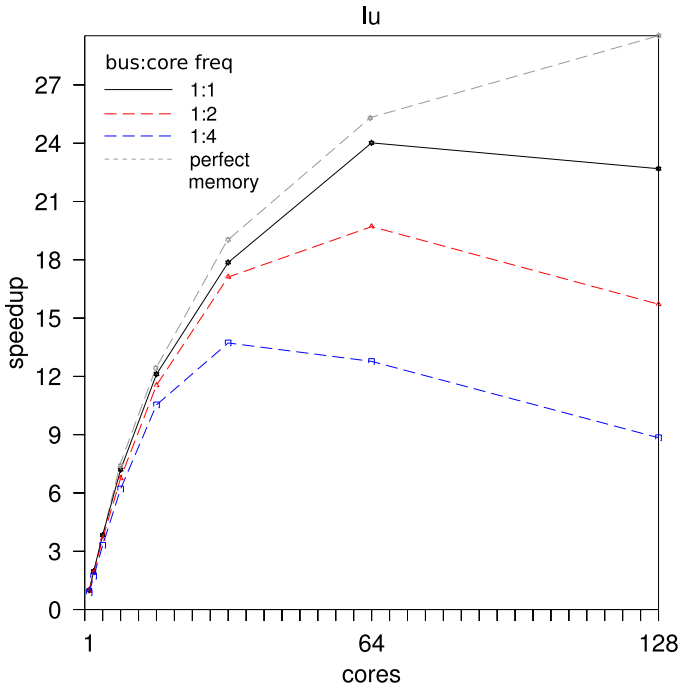


Figure 7.12: *Single bus CMP scaling - `lu`.*

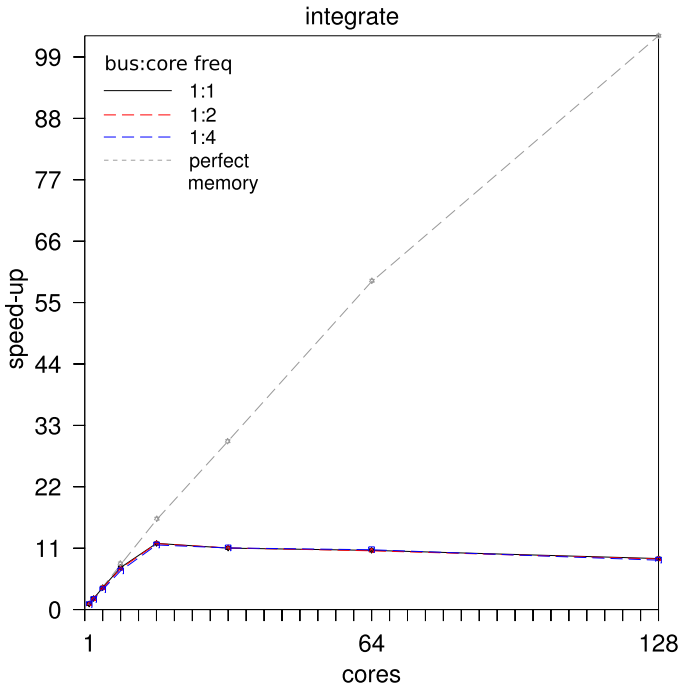


Figure 7.13: *Single bus CMP scaling - `integrate`.*

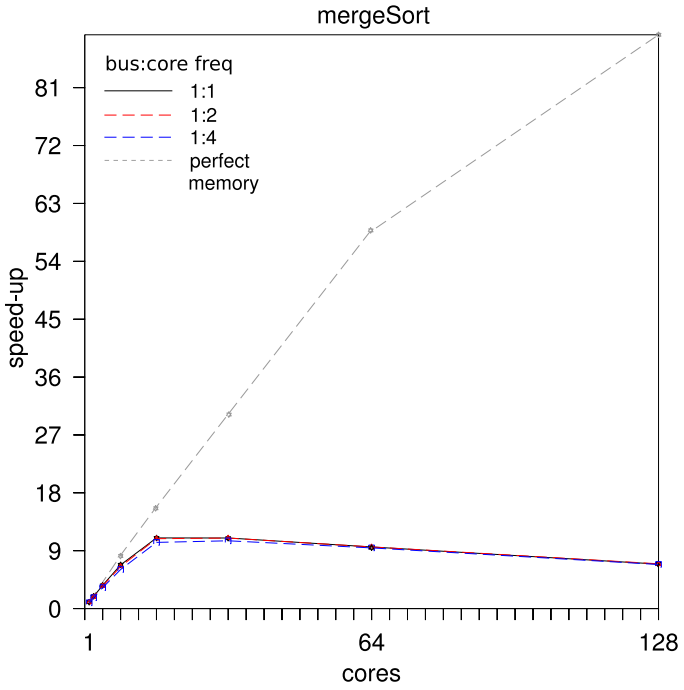


Figure 7.14: *Single bus CMP scaling - `mergeSort`.*

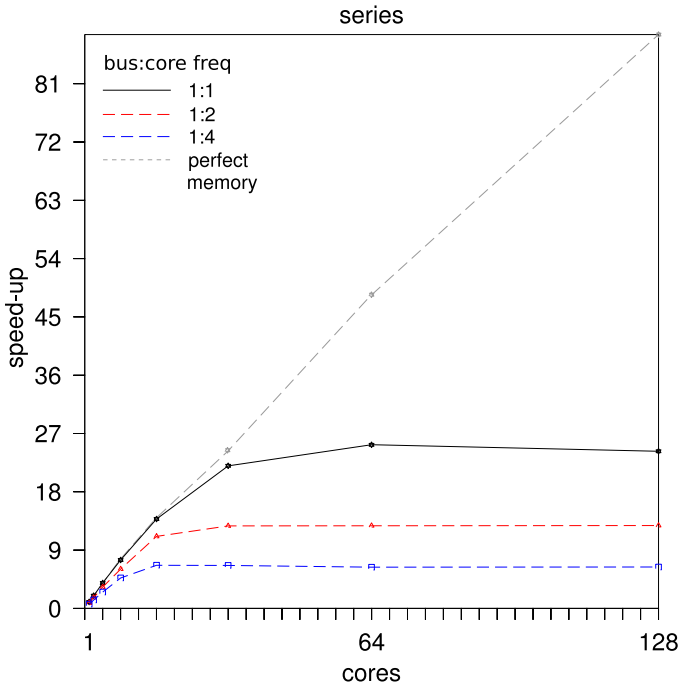


Figure 7.15: *Single bus CMP scaling - `series`.*

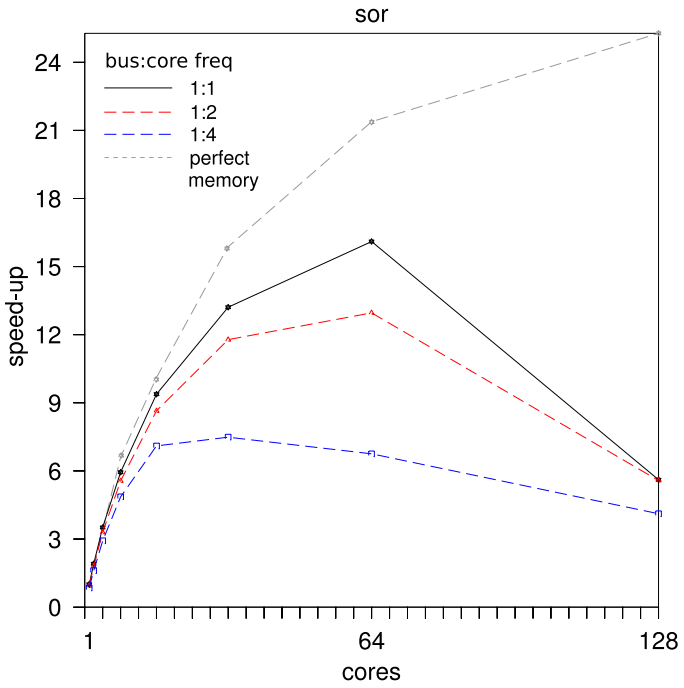


Figure 7.16: *Single bus CMP scaling - `sor`.*

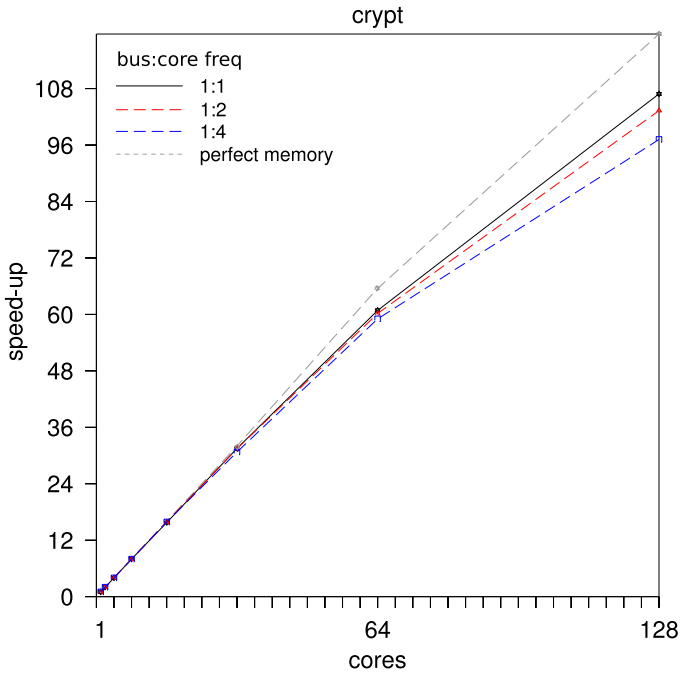


Figure 7.17: *Single bus CMP scaling -* `crypt`.

7.4.2 Wire Delay

Figure 7.18 shows the peak performance of each benchmark where the level 1 bus clock is set to $\frac{1}{2}$ and $\frac{1}{4}$ of the core clock speed, relative to the level 1 bus being clocked at the same speed as the core.

As might be expected, the peak performance drops for all of the benchmarks. This is due in part to a decrease in bandwidth, as the total number of transactions serviced by the bus in any given time period is reduced, and also due to an increase in the observed access latency to the L2 shared cache and correspondingly any bus serviced cache-to-cache transfers. In particular the maximum speed-up for the *diminishing* benchmarks, `matrixMult`, `lu`, `series` and `sor`, decreases by between 18 and 50% when the bus speed is halved, and between 42 and 73% when the bus speed is quartered. Most of the performance is lost during slower accesses made to the L2 shared cache, as the data sets used are sufficiently large to overflow the private level 1 caches. The *near-linear* benchmarks, `fibonacci` and `crypt`, are less effected, however, peak performance is still observed to drop by between 4 and 21%. The *limited* benchmarks, `jacobi`, `integ` and `mergeSort`, are marginally affected by wire-delay, this is because they are more fundamentally

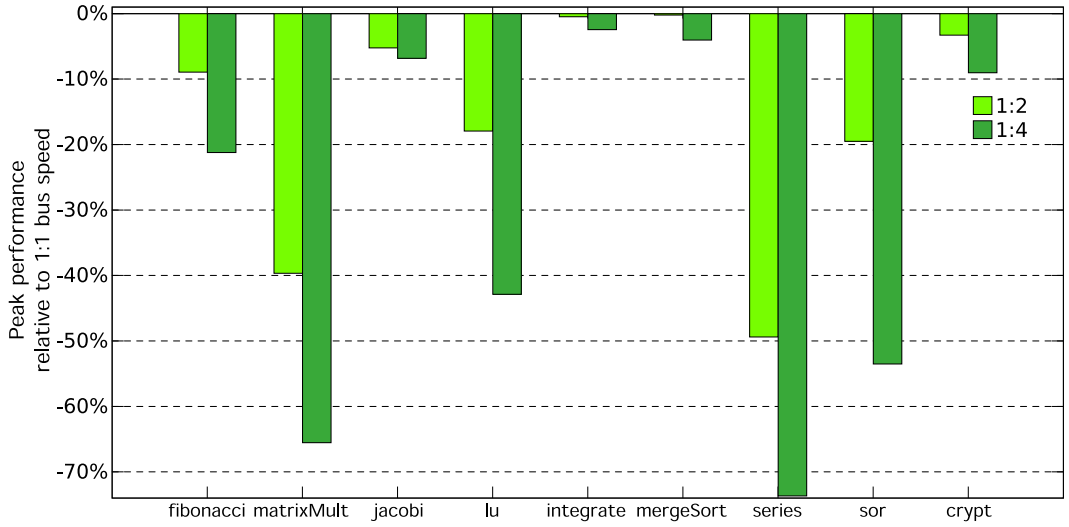


Figure 7.18: *Peak performance with the level 1 bus clock set at $\frac{1}{2}$ and $\frac{1}{4}$ of the clock speed, relative to the bus clocked at the same speed as the core.*

limited by memory saturation.

7.4.3 Bus Contention

Contention for the level 1 bus increases with the number of cores attached. The more cores, and hence more private caches that there are arbitrating, the longer any one cache is likely to have to wait until it is granted access to place a transaction on the bus. As mentioned previously, in Section 3.1.5, bus slots are granted in least recently used order to the L1 caches, with overall priority given to the L2 cache.

Figure 7.19, shows the peak and average utilisation of the level 1 shared bus which, for all of the benchmarks, increases with the number of cores. When the number of attached cores reaches 64 and 128, for the majority of the benchmarks, the average bus utilisation is well above 60% and the peak utilisation is over 90%. This accounts for the performance tail off seen in the *diminishing* benchmarks, as increasing the number of cores speeds up the processing of data, but the latency of access to that data also increases. For the *near-linear* benchmarks the bus utilisation is very low, `crypt` below 5% and `fibonacci` below 15%.

The corresponding utilisation of the channels between the L2 and L3 cache, Figure

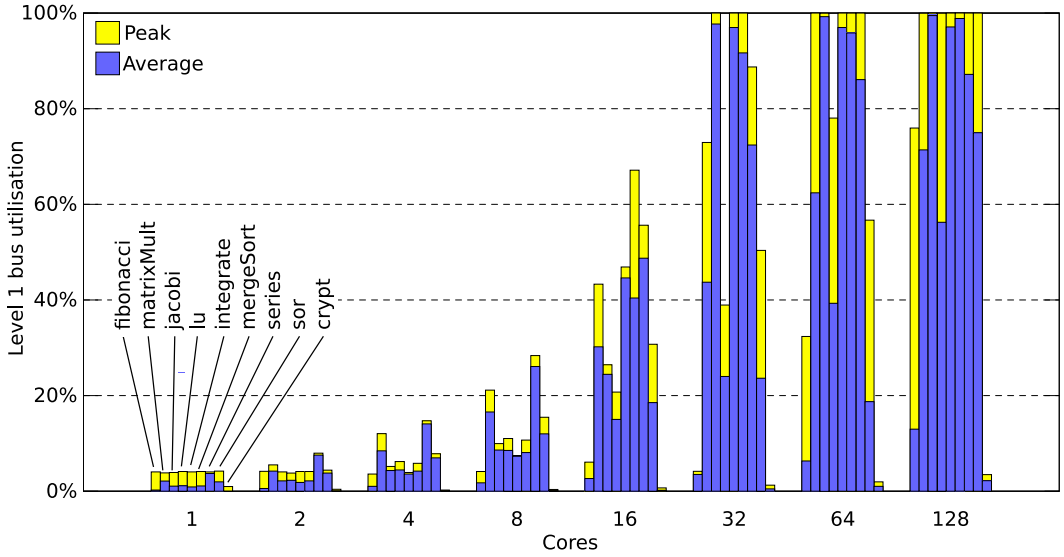


Figure 7.19: *Level 1 bus utilisation, average and peak during the benchmark phase, in a single bus CMP.*

7.20, never peak above 60% for any of the benchmarks, and for the majority the average utilisation is below 40%. This illustrates that the majority of contention on the L1 bus is for access to the shared level 2 cache or for cache-to-cache transfers.

7.4.4 Memory Saturation

Having looked at wire delay and bus contention, the third factor that can impact on the scaling performance of an architecture is memory saturation. This occurs when all of the cores are executing a data intensive benchmark leading to a bottleneck at the memory controller. This bottleneck leads to queue congestion and is observable as the number of transactions that receive nacks increases.

Such saturation is experienced by the *limited* benchmarks, `jacobi`, `integrate` and `mergeSort`. As the number of cores increases past 16 the average bus utilisation exceeds 90%, however over 80% of these transactions are negatively acknowledged, as shown in Figure 7.21. For this reason, none of the *limited* benchmarks are able to achieve greater speed-ups when the number of cores increases past 16.

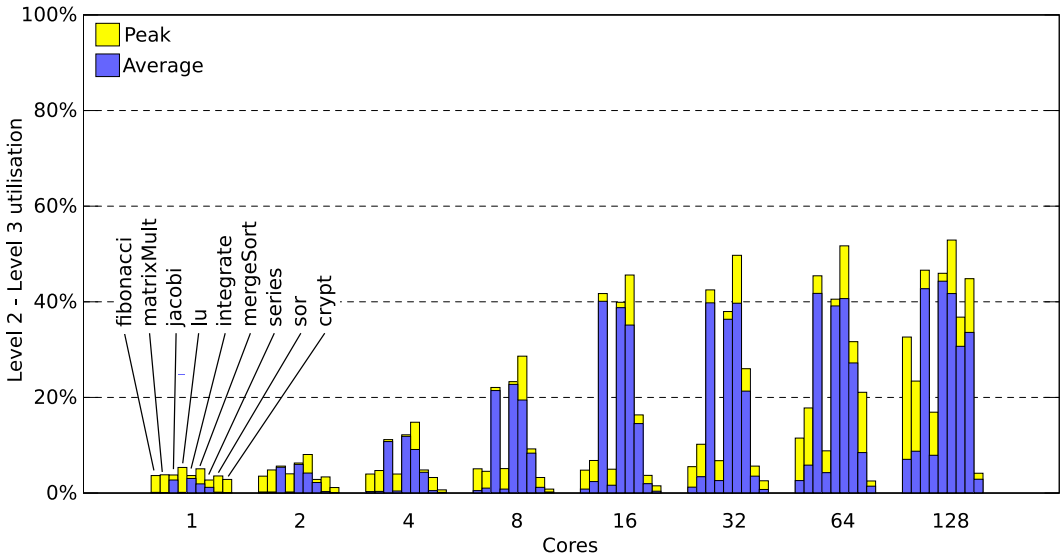


Figure 7.20: *Utilisation of bandwidth between the L2 and L3 caches, in a single bus CMP.*

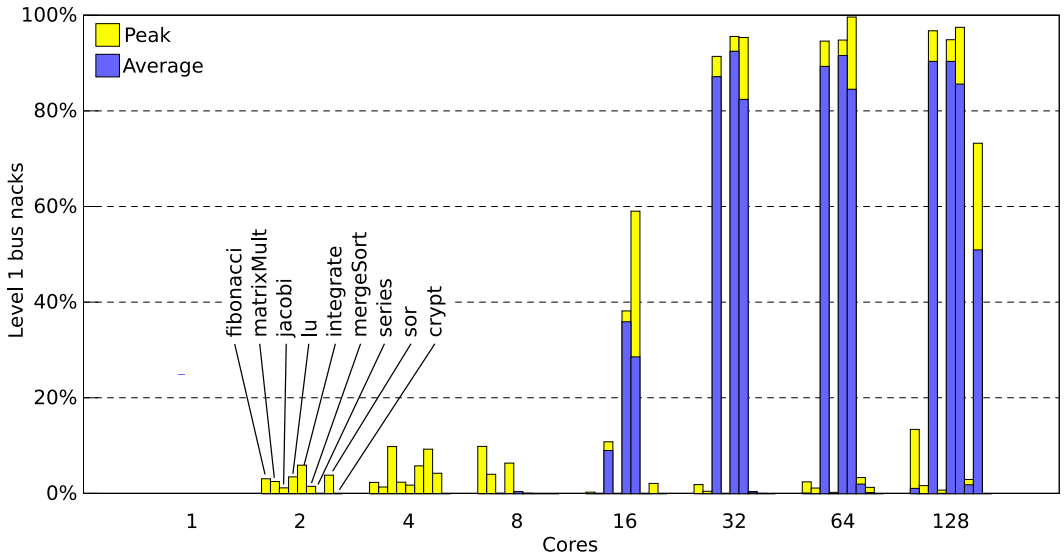


Figure 7.21: *Peak L1 bus nacks.*

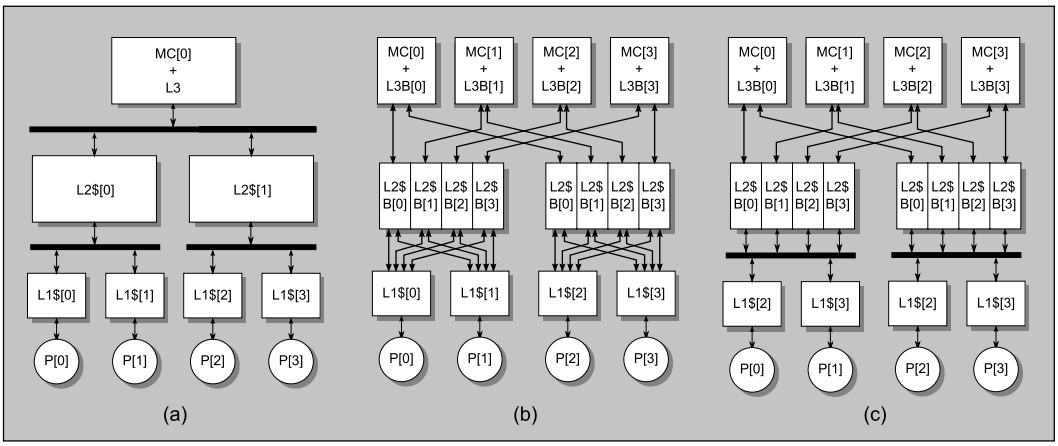


Figure 7.22: *Three chip multi-cluster (CMC) architectures are assessed, a) bus-tree, b) full crossbar and c) a bus-crossbar hybrid.*

7.5 Cluster Architectures

After illustrating the limitations of a single bus CMP, this section looks at dividing the processing cores into multiple on-chip clusters and additionally increasing the bandwidth to memory. Three architectures are simulated, illustrated in Figure 7.22, a chip multi-cluster (CMC) connected by a tree of buses, *bus-tree*, a CMC connected by crossbar switches, *full crossbar*, and a CMC connecting the cores with a shared bus, and connected at the cluster level by a crossbar switch, *bus-crossbar*. It should be noted that both the *full crossbar* and *bus-crossbar* architectures introduce three additional banked memory controllers, and therefore have four times the available bandwidth to memory.

Each architecture maintains the parameters presented in Figure 7.8. Where the architecture is divided into two and four clusters, the L2 cache size is also divided by two and four, such that the overall on-chip cache remains constant.

Clustering multiple-cores together and then building a hierarchy of clusters provides yet another level of abstraction at which to build a many-core architecture. The division into clusters also reduces the number of cores being serviced by any one shared cache, and so either the bus connecting them can be shorter and clocked at higher speeds or a smaller crossbar structure can be used. A disadvantage to clustering is that an additional level of latency is introduced when multiple cores in different clusters frequently share data.

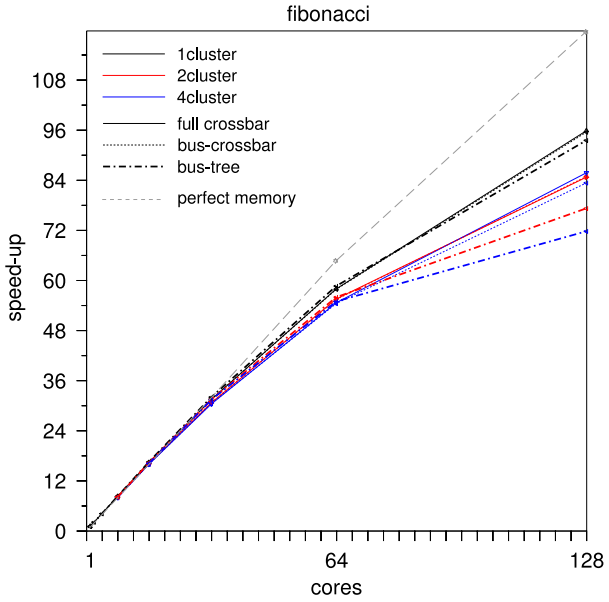


Figure 7.23: *CMC scaling - `fibonacci`.*

7.5.1 Speed-up

For each of the three architectures the same benchmarks were again executed to assess their scalability, the benchmark scaling graphs are presented in Figures 7.23 – 7.31. Each architecture was configured in a single cluster configuration, which, for the bus-tree CMC, is analogous to the single-bus CMP, and two and four cluster configurations. The clusters are simulated in symmetrical configurations, such that the total number of cores and shared cache is divided equally between the clusters.

Figure 7.32 shows the peak performance achieved by each of the cluster architectures for each benchmark normalised to the peak performance of the single bus CMP architecture, with the bus speed set at the same speed as the core frequency. This scenario is perhaps unfair because a single bus connecting 128 cores at the core clock speed is considered infeasible, however, it does provide a best-case single bus to compare each cluster architecture configuration against.

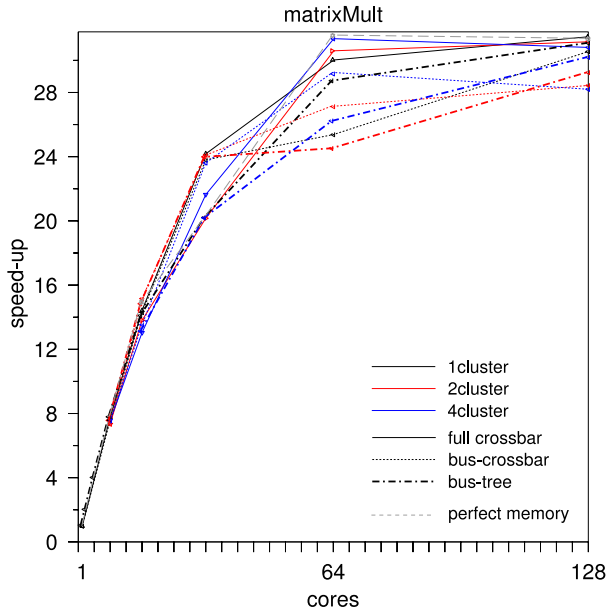


Figure 7.24: *CMC scaling - `matrixMult`.*

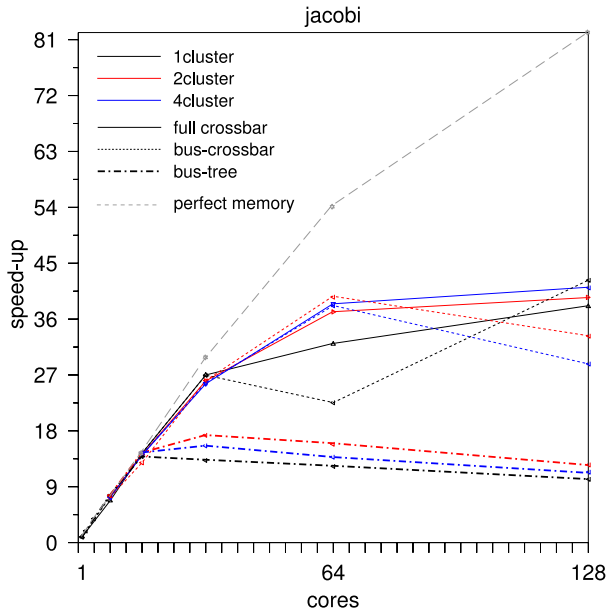


Figure 7.25: *CMC scaling -`jacobi`.*

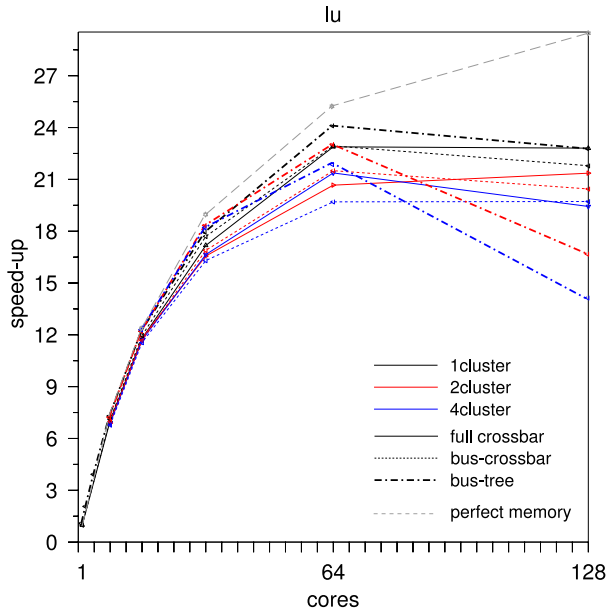


Figure 7.26: *CMC scaling - `lu`.*

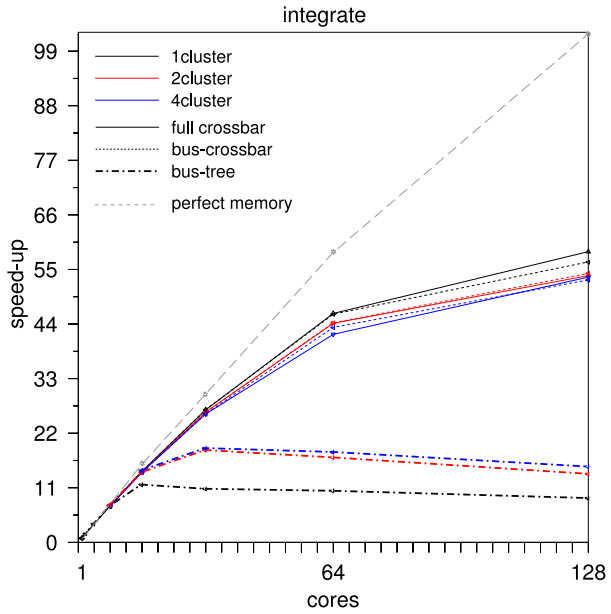


Figure 7.27: *CMC scaling - `integrate`.*

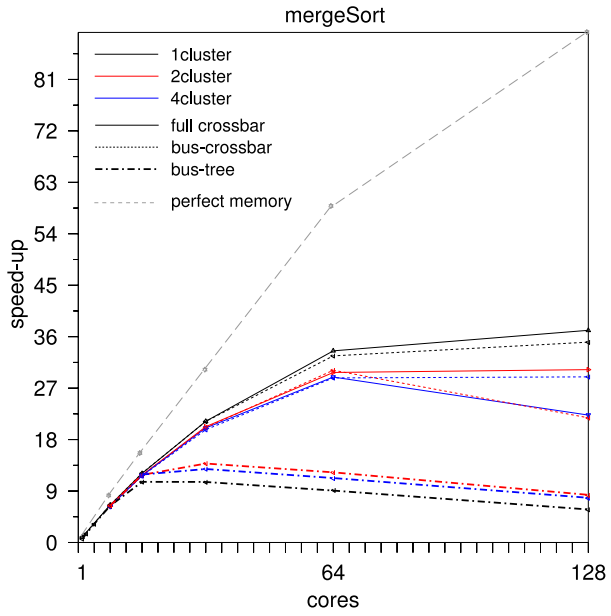


Figure 7.28: *CMC scaling - `mergeSort`.*

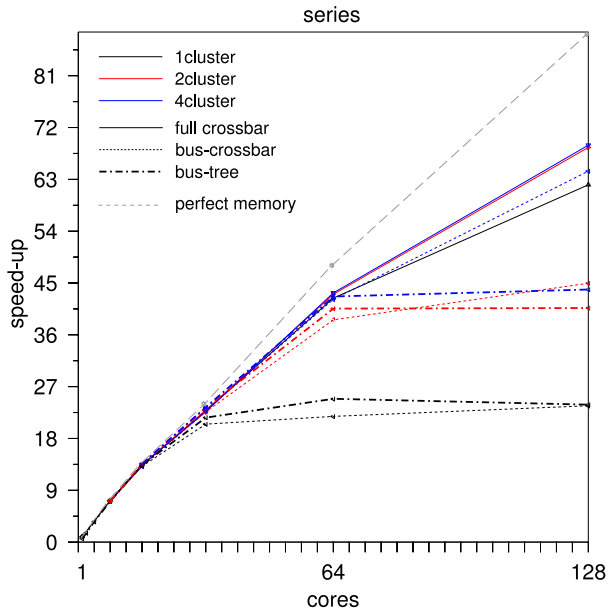


Figure 7.29: *CMC scaling - `series`.*

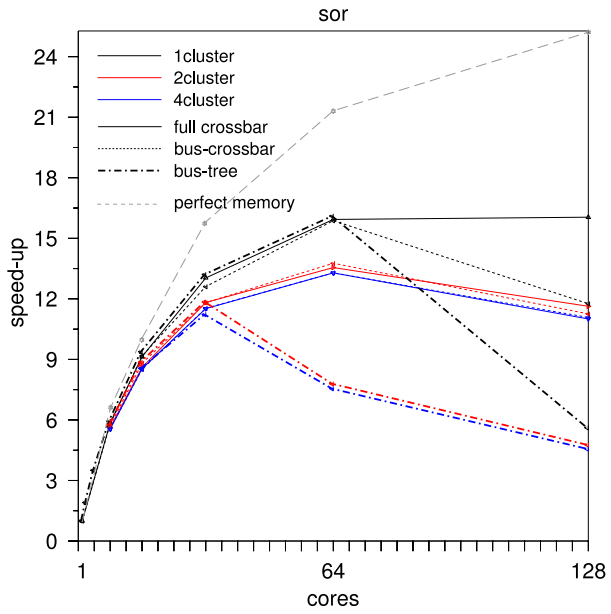


Figure 7.30: *CMC scaling - `sor`.*

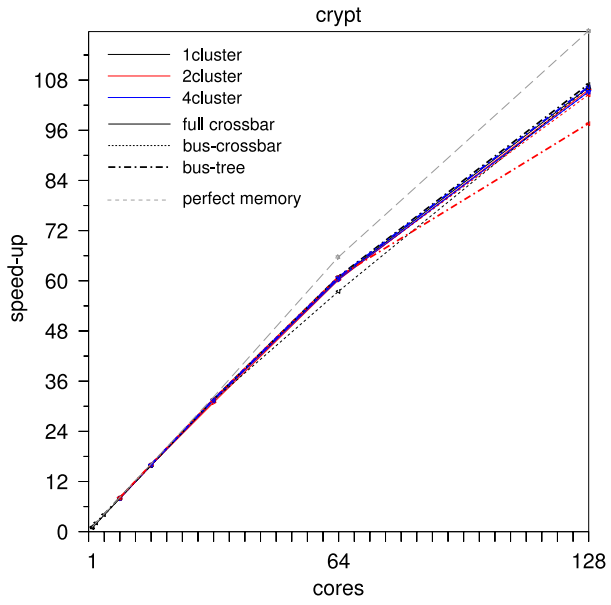


Figure 7.31: *CMC scaling - `crypt`.*

7.5.2 Bus-Tree Cluster

Looking first at the performance of the *bus-tree* cluster architecture, Figure 7.22 (a), the general trend is that the introduction of additional clusters decreases the peak speed-up for most of the benchmarks. As the threads of each benchmark are spread across multiple clusters, and are sharing data, the additional latency in accesses to this data is impacting on the performance. The benchmarks that suffer from this increased latency the most are `fibonacci` and `lu`. The peak performance is reduced by 18 and 27% for two clusters and 24 and 32% for four clusters respectively. These reductions, however, are compared to the fastest clocked bus, and referring back to Figure 7.18, `fibonacci` performance is reduced by 21% when wire delay reduces the bus speed to $\frac{1}{4}$ and `lu` performance is reduced by 42%. Taking these wire delay reductions into account for the single cluster architecture the disparity is reduced to to -6% for `fibonacci` and +10% for `lu` on a four cluster machine.

Four of the benchmarks, `jacobi`, `integrate`, `mergeSort` and `series` benefit from the division into multiple clusters. Referring back to Figure 7.19 it is these four benchmarks that have peak and average bus utilisation above 80% when the number of cores is either 64 or 128. The addition of multiple clusters reduces the access contention on the level 1 bus, as there are fewer cores attached, and correspondingly reduces the access latency to the level 2 shared cache. For `series` and `integrate`, reduced level 1 bus contention sees the peak performance increase by 1.5 and 1.85 times. Prior work by Nayfeh *et al.* [116] looking at the effects of clustering in small-scale shared-memory multiprocessors also showed a benefit in performance due to a reduction in bus contention.

7.5.3 Crossbar Cluster

The second architecture presented, the *crossbar* architecture, Figure 7.22 (b), is simulated with a full $n \times 4$ crossbar, where n is the number of cores in each cluster, and 4 is the number of banks in the shared L2 cache. Each L2 cache is connected by a $(c \times 4) \times n$ crossbar, c being the number of clusters, to an L3 cache which is also divided into 4-banks. Each L3 bank is connected to a separate memory controller. The addresses are divided into the four banks at the cache-line, 32-byte, granularity using the offset and mask show in Equation 7.6.

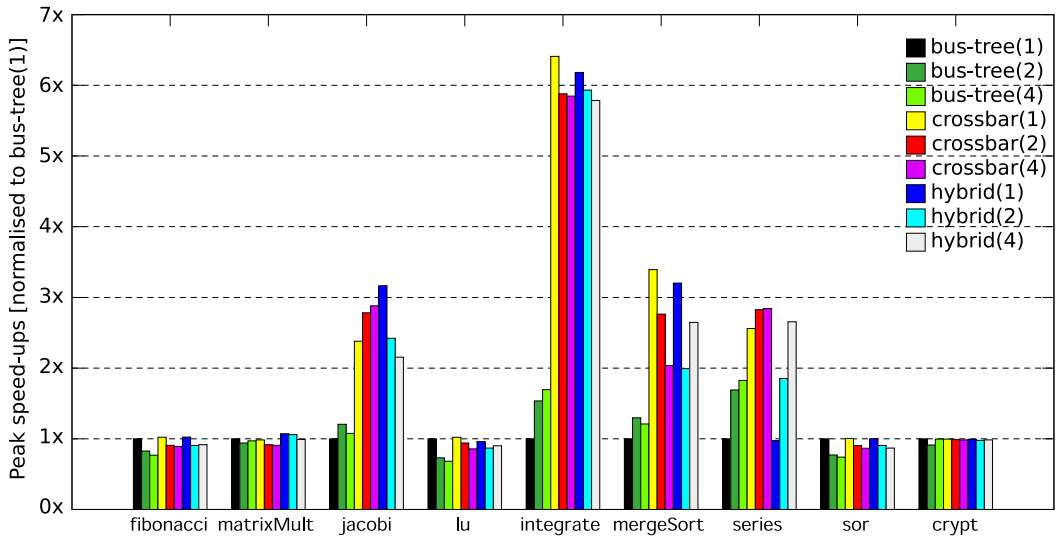


Figure 7.32: *Peak speed-up achieved, by the three CMC architectures, normalised to the performance of bus-tree(1)(the single bus CMP).*

$$bank = ((address >>> 5)\ \&\ 3) \tag{7.6}$$

Dividing the L2 and L3 caches into banks and increasing the number of memory controllers to 4, provides a fourfold increase in the available memory hierarchy bandwidth. Additionally the full crossbar between the cores and the banks of the L2 cache provides an approximately fourfold increase in the number of coherence transactions that can be processed[3]. In the single cluster configuration, this architecture closely resembles the Niagara architecture [81] with the addition of L3 caches.

This has a significant impact on the scaling of the benchmarks `jacobi`, `integ`, and `mergeSort` which, as mentioned previously in Section 7.4.4, are limited in a single bus CMP due to memory bandwidth saturation. In particular the `integ` peak speed-up is increased by over 6 times, shown in Figure 7.32. This is due to both a combination of a fourfold increase in the memory bandwidth but also the reduction in level 1 bus contention. `mergeSort` and `jacobi` also see greater than two-fold increases in peak performance.

[3]The increase in coherence transactions that can be processed across the crossbar peaks at four times the amount across the bus, however in certain cases additional queueing is necessary. In particular to achieve a broadcast, for invalidation say, requires the ability to send a signal to all cores holding the line, or failing that stall until the relevant channels inside the crossbar are free.

The additional latency associated with sharing data across clusters decreases the performance, by as much as 40%, for `mergeSort`, as the total number of cores is divided into four clusters.

7.5.4 Hybrid Bus-Crossbar Cluster

The third simulated cluster architecture, the bus-crossbar 7.22 (c), connects the private L1 caches to the four banks of a shared L2 cache using a single bus, and connects the L2 caches to the L3 cache banks and memory controllers using full crossbar switches. The bus arbitration is modified, so that while priority is generally given to the L2 cache, each bank is selected in least recently used (LRU) order.

The hybrid architecture was simulated for two reasons. First, it enables distinction between the benefit of additional memory bandwidth and that of increased transaction throughput. By maintaining a bus to connect the private L1 caches to the L2 caches, bus contention in the absence of memory saturation can be more readily observed. Secondly, the cost of a full crossbar interconnect in terms of area has been shown to reduce chip real-estate otherwise available for additional cores or cache [84]. The performance achieved using the hybrid bus-crossbar architecture can therefore be compared to that of the full crossbar architecture.

The `series` benchmark running on the hybrid bus-crossbar architecture clearly demonstrates that even though the memory bandwidth is quadrupled beyond the L2 cache, the primary limiting factor to performance gains over the single bus CMP is contention between the L1 caches and the shared L2 cache. Dividing the number of cores between two clusters provides a 1.8 times increase in peak performance, and division into four clusters a 2.7 times increase. Even though the primary limiting factor is bus contention, the hybrid architecture provides an additional 80% speed-up over the clustered bus-tree architecture.

In general the peak performance achieved using the hybrid bus-crossbar cluster outperforms that of the bus-tree architecture and is within 10% of that of the full crossbar architecture. Assuming that larger caches could be added in the absence of full crossbars between the L1 and L2 caches, the performance discrepancy between a *crossbar* and *hybrid* CMC architectures will likely drop further. Where

the performance drops, as the cores are divided amongst additional clusters, for both the full crossbar and the hybrid bus-crossbar architectures the decrease is less than that of the respective decreases in the bus-tree architecture.

7.6 Locality Aware Task Distribution

As outlined in Chapter 6, dividing the total number of cores in a CMP architecture into multiple clusters, where each cluster contains at least one shared level of cache, introduces an additional form of locality. Cluster locality, the notion of sharing data internally within a cluster, can be exploited to utilise the shared level of caches within each cluster efficiently. To investigate the benefits of exploiting this, the locality-aware task distribution mechanism described in Section 6.2.4 was implemented within the cycle-level simulation platform.

Three experiments were used to assess the benefits of locality-aware task distribution with respect to *synchronisation*, *isolation* and *affinity*. A simple framework implemented within the Jamaica port of the Jikes RVM was developed to allow software to assign a cluster affinity to each application thread generated during the execution of a benchmark. These application threads are either distributed using Jamaica's token ring task distribution mechanism, as described in Section 3.1.3, or distributed based on the cluster affinity assigned to them. Cluster affinity is assigned to a Java thread by calling the method `setClusterAffinity`, as listed in Figure 7.33. This call instructs the virtual machine to pass an affinity value along with the token request `TRQ` instruction whenever the thread is scheduled on to another `VM_Processor`. On execution of the `TRQ` instruction the hardware attempts to locate a token inside the required cluster. For each set of experiments the simulated architectures were configured to assess the effects of locality-aware task distribution over a range of both clusters and cores per cluster.

7.6.1 Synchronisation Locality

To assess the cost of synchronisation in a chip multi-cluster architecture the low-level `barrierBench` benchmark is used. Two cluster architectures are assessed, the bus-crossbar *hybrid* CMC and the full *crossbar* CMC. For each architecture

```
int level = 2; //look for clusters below the L3 cache.
int numberOfClusters = VM_Scheduler.getClusters(level);

for(int i = 0; i < 16; i++) {
   int clusterId = i % numberOfClusters; //divide the threads amongst the clusters
   th[i] = new Thread(benchmarkRunner[i]);
   VM_Scheduler.setClusterAffinity(th[i], clusterId);
}
```

Figure 7.33: *Setting a cluster affinity to Java threads.*

the task distribution mechanism is either configured to distribute tasks based on the *cache-distance* metric passed through a `TRQ` instruction, *locality-aware* distribution, or to distribute tasks to any idle contexts within the system, *default* distribution. For each simulation several instances of the `barrierBench` benchmark are invoked, such that the number of instances is equal to the number of clusters. Figure 7.34 shows the results from these experiments.

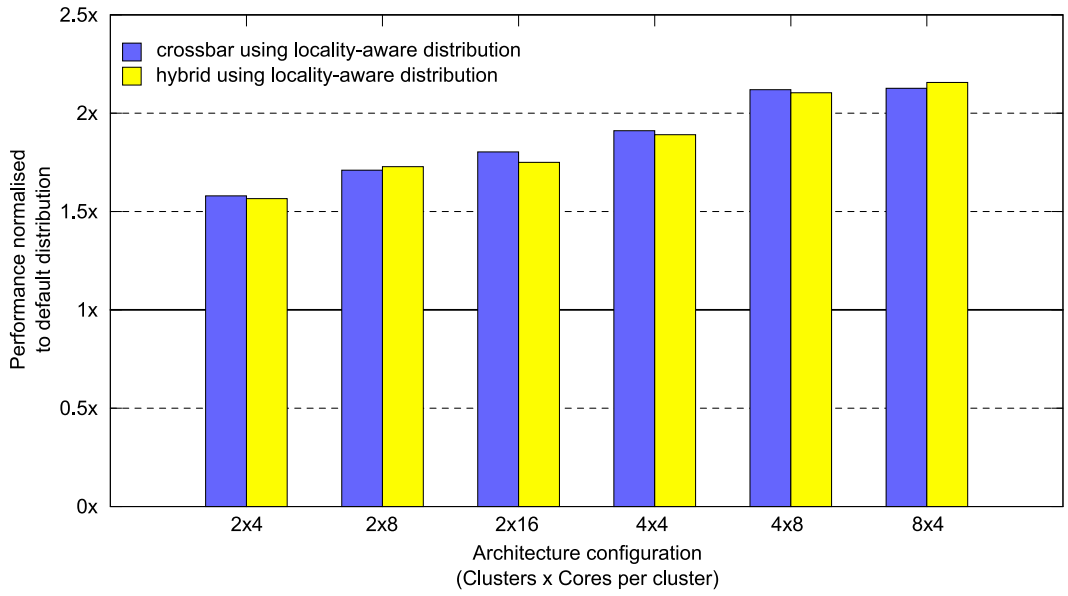


Figure 7.34: *The effect on overall performance running multiple instances of the `barrierBench` benchmark across multiple cluster architectures using locality-aware distribution.*

Restricting each `barrierBench` instance into a single cluster increases the overall performance by a factor of up to 2. This improvement, using the locality aware distribution scheme, is achieved as almost all synchronisation in the benchmark occurs within a cluster and so the latency of access is reduced.

7.6.2 Application Isolation

A potential benefit of dividing the on-chip caches and cores into clusters is that applications can be isolated within a single cluster. Potentially this can improve cache performance as all cores within a cluster can benefit from locality of application code and data. The effects of deconstructive sharing from other application threads can also be eliminated. To assess the effect of isolating applications the *hybrid* and *full-crossbar* CMC architectures were once again used in simulation. For each cluster in the architecture a separate instance of the `sor` benchmark is invoked, each thread generated by the application is restricted, through the *locality-aware* distribution mechanism, to run inside the cluster of the initial `sor` application thread. The work of each `sor` benchmark is divided into a number of threads, such that there is a thread for each core inside the cluster. Figure 7.35 shows the results from these experiments.

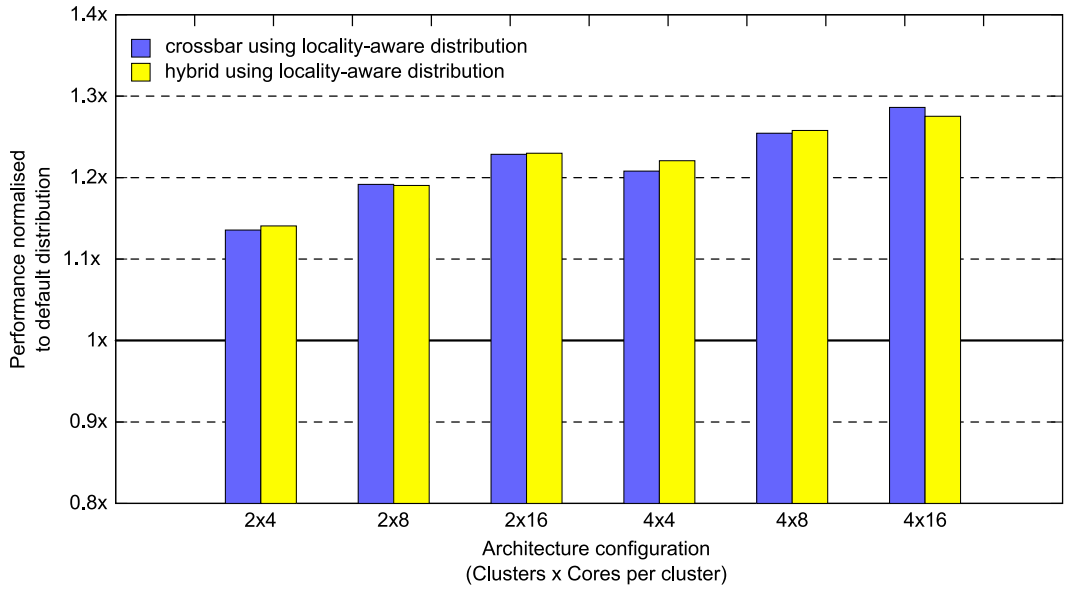


Figure 7.35: *The effect on overall performance running multiple instances of the* `sor` *benchmark across multiple cluster architectures using locality-aware distribution.*

As was shown in Section 7.6.1, isolating applications into clusters can improve performance as the shared synchronised data was confined in each clusters shared L2 cache. By confining each instance of the `sor` application into a separate cluster, and each instance's threads within that same cluster, data sharing in the L2 caches is improved and inter-cluster coherence traffic is reduced. This improves

the performance of multiple `sor` applications running on the CMC architectures by up to 1.29 times.

7.6.3 Application Restructuring

Although isolating an application within a single cluster in a CMC architecture can improve performance, by increasing the cache efficiency and reducing access latencies to shared data, there will be occasions when a single application should be executed across multiple clusters or indeed the whole chip to maximise performance. As shown in Figure 7.32 there is an associated decrease in performance for most benchmarks as the number of clusters is increased. This performance decrease is mainly caused by the additional latency when accessing shared data across clusters. However, when careful consideration is given to the distribution of work inside a benchmark the latency of accessing shared data across multiple clusters can be significantly reduced. Most data sharing between threads can be confined to the cluster they share, reducing the access latency and increasing the performance.

To demonstrate this the `sor` benchmark is restructured to minimise sharing of data between threads on different clusters. The initial `sor` algorithm divides the grid, over which the successive over relaxation is calculated, into equal sized strips such that one strip is given to each worker thread, see Figure 7.36 (a). During each step of the algorithm, either the red or black elements are calculated by reading the values of the four nearest neighbours and the element itself. Given a sufficiently large sized grid, good parallelism can be achieved. Each thread is only modifying the black (red) elements in one strip and the value is calculated from the four nearest red (black) neighbours which are not being modified.

In a CMC architecture a naïve distribution may place adjacent strips into separate clusters, Figure 7.36 (b). Accessing the updated values of neighbouring elements for all strips requires communication across the top level network, which has increased latency and lower bandwidth. However, by structuring the application such that, as far as possible, adjacent strips remain on cores within the same cluster, Figure 7.36 (c), access to data across the slowest communication paths in the architecture is reduced.

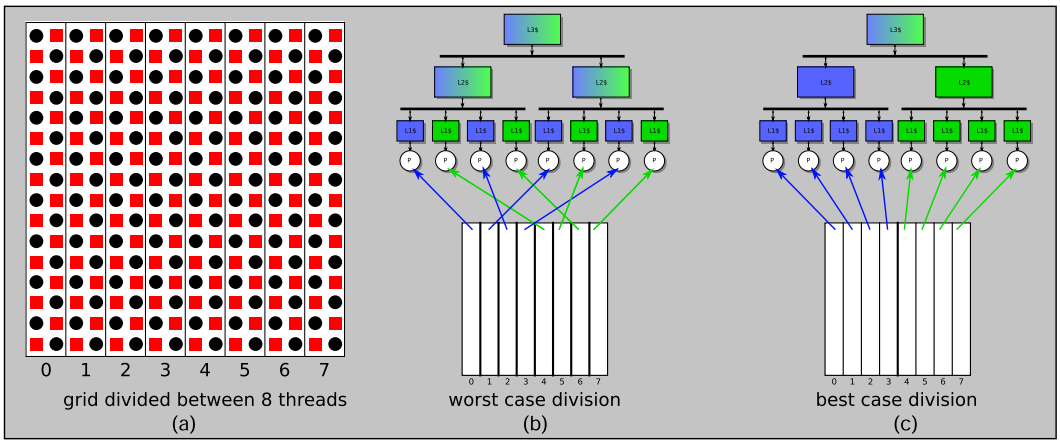


Figure 7.36: *The* `sor` *benchmark: (a) the grid is split into strips each of which is distributed to a worker thread (b) pathological distribution can see all threads having to communicate across the higher latency bus to access data in another thread. (c) optimal division of the adjacent threads into cores within the same cluster. Communication across the top-level bus is restricted to the overlapping data shared between threads 3 and 4.*

Figure 7.37 shows the relative increase in performance using locality-aware thread distribution, compared with using the default distribution scheme. The locality-aware scheme first attempts to distribute threads to cores that are idle within a defined cluster. If a core within the defined cluster is not found the default distribution is employed. In the default distribution threads are distributed to any idle core, or in the presence of no idle cores are executed on the context attempting the distribution.

For all of the cluster configurations simulated, the locality-aware thread distribution scheme decreases the total execution time of the benchmark. The relative performance of the scheme increases as the number of cores per cluster is increased. This is an expected result, as increasing the cores, and hence the threads in the benchmark, also increases the likelihood in the default scheme that adjacent threads, in the algorithm of the `sor` benchmark, will be distributed to separate clusters. Data sharing between clusters suffers from an increased communication latency and reduced bandwidth and becomes a performance bottleneck if significant.

Table 7.3 lists the total coherence traffic, the total number of four-phase transactions and the average thread distribution distance[4] for the inter-cluster network

[4]The average sharing level distance that threads are distributed to. The average is calculated

Configuration		Total Transactions			Total Four-phase Transactions			Average Distribution Distance			Relative Speed-up
Clusters	Cores	Default	Locality	Diff.	Default	Locality	Diff.	Default	Locality	Diff.	using locality scheme
2	4	4531324	1095510	4.13×	245791	104395	2.35×	1.12801	1.03065	1.09×	1.08×
2	8	5031430	947755	5.30×	291240	109961	2.64×	1.05526	0.964409	1.09×	1.11×
2	16	5724974	1140433	5.02×	636814	126099	5.05×	0.950121	0.870201	1.09×	1.15×
2	32	6120999	927995	6.59×	1057511	134613	7.85×	0.827252	0.740046	1.11×	1.26×
4	2	5965035	2030906	2.93×	301981	180473	1.67×	1.41104	1.0204	1.38×	1.06×
4	4	7111354	1811588	3.92×	405490	191038	2.12×	1.17029	1.00621	1.16×	1.07×
4	8	7557023	1374522	5.49×	808084	168685	4.79×	1.00467	0.921874	1.08×	1.15×
4	16	12295826	2599454	4.73×	1934749	310533	6.23×	0.905064	0.797526	1.13×	1.27×
8	2	7592591	2637468	2.87×	466029	309951	1.50×	1.42537	1.00757	1.41×	1.05×
8	4	9001945	1979883	4.54×	918022	233509	3.93×	1.14582	0.94696	1.20×	1.12×
8	8	13122791	2818096	4.65×	2102581	387041	5.43×	0.958656	0.801404	1.19×	1.29×

Table 7.3: *Locality-aware task distribution: reduction in coherence traffic, four phase transactions and the average distribution of threads.*

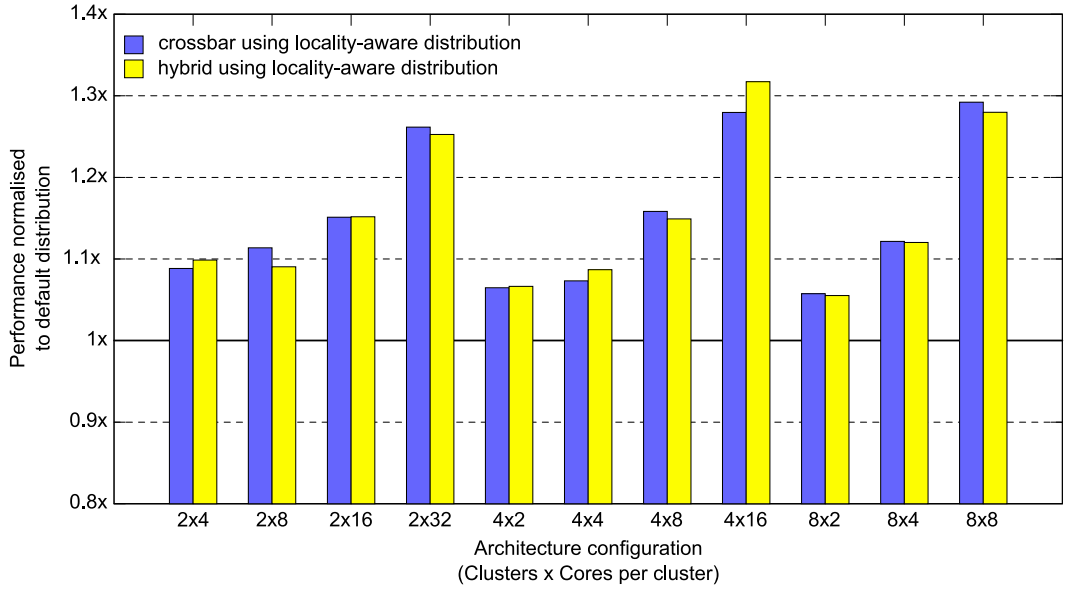


Figure 7.37: *The effect on overall performance running multiple instances of the* `sor` *benchmark across multiple clusters using locality-aware distribution.*

in the full-crossbar cluster architecture, shown in Figure 7.22 (b). The results are shown for both the default and the locality-aware distribution scheme. The difference, *Diff.*, column shows the reduction achieved using the locality-aware scheme.

Using the locality-aware distribution, a reduction in total coherence traffic and four phase transactions is observed on the inter-cluster bus for all configurations. Correspondingly a reduction in the average thread distribution distance is observed at each processor on executing a `TRQ` instruction. These two factors account for the improved performance using the locality-aware scheme.

7.7 Chip Multi-Cluster Design Considerations

The previous section presented results showing that performance increases are achievable through restructuring of an application and use of a locality aware thread distribution scheme. However, Section 7.5 illustrated that increasing the number of clusters often leads to decreased performance when data is frequently

over all `TRQ` operations. A value of 0 is added to the cumulative value if the thread remains in the same context, 1 if the threads remain in the same processor, 2 in the same cluster, 3 outside the cluster. Further explanation can be found in Chapter 6.

shared across clusters. Equally, for some configurations where the number of cores is small, a CMC architecture may actually perform worse than a single bus CMP architecture.

In order to gain some insight into the design considerations required for utilising CMC architectures, the performance of a single cluster bus-crossbar architecture, see Figure 7.22 (c), is compared to the performance of a set of full-crossbar and hybrid bus-crossbar CMC architectures, see Figure 7.22 (b) and (c). Each architecture is configured with an equal number of on-chip memory controllers, and an equal amount of total cache. For the single cluster bus-crossbar architecture, the inter-core bus is clocked at $\frac{1}{4}$ of the core clock.

Single Bus CMP ($\frac{1}{4}$ bus) cycles	Configuration		Hybrid		Full Crossbar	
	cluster	cores	cycles	speed-up	cycles	speed-up
180520129	2	4	196444747	0.92×	196803387	0.92×
104599844	2	8	90385345	1.15×	91011825	1.15×
88412428	2	16	53418809	1.66×	54501857	1.62×
92687948	2	32	26846465	3.45×	26732865	3.47×
180520129	4	2	305976453	0.59×	306313453	0.59×
104599844	4	4	116654491	0.89×	116536067	0.90×
88412428	4	8	56110859	1.57×	54547371	1.62×
92687948	4	16	43739515	2.11×	39954355	2.32×
104599844	8	2	171528459	0.61×	170192259	0.61×
88412428	8	4	65979083	1.34×	66460291	1.33×
92687948	8	8	40100163	2.31×	39831179	2.33×

Table 7.4: *Performance comparison between the locality-aware optimized* `sor` *benchmark running on the full crossbar CMC architecture, and the* `sor` *benchmark running on a single inter-core bus CMP, with 4 memory controllers: a single cluster version of the bus-crossbar hybrid (see Figure 7.22 (c)).*

The `sor` benchmark is executed on each architecture. The results presented in Table 7.4, compare the execution of the optimized `sor` benchmark on each clustered architecture against the execution on the single cluster bus-crossbar hybrid architecture, containing the same total number of cores.

Where the total number of cores is 8 or 16 the single bus architecture generally out performs the CMC architecture, for the `sor` benchmark. There are several design considerations here. Access latency to shared data is uniform for all of the cores in the single bus architecture. Additionally bus contention is less than 40%, see Figure 7.19, and is therefore not a limiting factor. The CMC architectures perform poorly as latency to shared data is increased when sharing occurs across

clusters.

For 32 and 64 cores, however, the CMC architectures always out perform the single bus architecture. The increased latency of access to the level 2 cache, reduced bus bandwidth and high bus contention levels, when the bus speed is reduced, impact on the performance attainable from the `sor` benchmark in the single bus CMP.

Recent studies have shown that there is also a need to consider the cost in area of on-chip interconnects [84] and the efficiency of cache configurations [72, 69] when trying to optimise CMP performance.

7.8 Summary

This chapter has evaluated, through cycle-level simulation, the coherence protocol introduced in Chapter 4 implemented using the hardware support introduced in Chapter 5. The architecture and protocol have been exercised using 10 representative parallel benchmarks. A single bus CMP was simulated and demonstrated that wire delay and bus contention both inhibit scaling in large, greater than 32 core, configurations.

Three CMC architectures were simulated, demonstrating the protocol and architecture's capability of maintaining coherence across multiple clusters. The architectures are able to exploit more parallelism from the benchmarks than the single-bus CMP. The architectures reduce the effect of wire delay by decreasing the span of the inter-core bus or crossbar. Bus contention is also reduced as the number of cores connected to each inter-core bus, or to the banked cache in a crossbar, decreases. The cluster architectures also demonstrated the necessity for multiple memory controllers in order to avoid memory saturation.

Finally a locality-aware thread distribution scheme, introduced in Chapter 6, was demonstrated to reduce the cost of synchronisation and deconstructive cache sharing by isolating separate applications inside separate clusters. Furthermore restructuring the `sor` benchmark demonstrated the ability to increase performance by introducing simple locality-aware optimisations into the benchmark. These optimisations enable up to 3.4 times improvement in performance, when

executing a restructured benchmark on a CMC architecture, over that of a wire delay limited single bus CMP.

CHAPTER 8

Conclusions

As the number of transistors integrated onto a silicon chip continues to grow, so the potential to incorporate more processing cores becomes a reality. Current CMP architectures contain a relatively small number of processing cores, up to eight, and hardware support for up to 32 concurrent threads. It is realistic, then, to expect that a trend of increasing the number of processing cores will emerge in an attempt to maximise both the power and performance efficiency of the increasing single chip transistor budget.

Currently, however, there is a limited understanding of the effects that incorporating tens of processing cores will have on the cache, memory and interconnect within a single chip architecture. This thesis represents an investigation into the effects that scaling, into the hundreds of cores, has on cache efficiency, interconnect utilisation and memory saturation. With the limits imposed through wire-delay in modern process technologies, and the need to bridge the growing design complexity gap, a chip multi-cluster (CMC) architecture is proposed as a viable design solution.

The caches in the CMC architecture maintain coherency using a multi-level cache coherence protocol, presented in Chapter 4 and hardware extensions introduced

in Chapter 5. An extension to the instruction set architecture, Chapter 6, enables software optimisations that are able to distribute work across a CMC architecture to exploit locality.

8.1 Contributions

The thesis outlined five contributions to knowledge:

A Multi-level Coherence Protocol

A protocol capable of maintaining shared memory cache coherence over multiple levels of on-chip shared cache was presented. The protocol is based on four-phase transactions; request, action, reaction, response. It generalises sufficiently to maintain coherence across both bus and crossbar interconnects. An explicit pending state in the protocol is used to prevent unnecessary coherence traffic propagating onto lower level buses while four-phase transactions are in flight.

Hardware Support for Multi-Level Coherence

Cache hardware required to support a multi-level coherence protocol was presented. In particular a core- and memory-bound queueing systems, necessary at each shared cache, were outlined. The addition of multiple levels of shared cache introduces flow control and deadlock issues into the cache hierarchy. Using dual-channel, sinkable and non-sinkable queues, deadlock is avoided by breaking circular chains of dependence. A novel passive and active queueing mechanism was presented that allows reordering of non-sinkable messages to prevent head of queue blocking.

Locality-Aware Task Distribution

An extension to the instruction set architecture was introduced allowing software to exploit cache locality by controlling the affinity of distributed tasks. The extension allows software to distribute threads to a core anywhere within the

architecture based on a cache-distance metric and token identifier. The token identifiers are used to encode the cache-distance metric providing a simple method by which threads can be distributed across the chip.

CMP/CMC Simulation Platform

A simulation platform was developed in order to undertake the work contained within this thesis. The simulation platform is capable of simulating CMP and CMC architectures, interconnected by bus or crossbar interconnects, containing multiple cache levels, and many hundreds of cores or contexts.

Cycle-level implementations of the coherence protocol, the cache hardware support and the locality aware task distribution scheme were incorporated into the simulation platform to enable the experimental analysis undertaken in this thesis.

Fully Cache Coherent, Multithreaded Study

Finally, while other research has explored the area of large-scale CMP architectures [72, 69] these studies have focused on exploration and trade-offs specifically in the cache design space. The studies were based on statistical analysis using synthetic trace-driven simulations. In contrast the investigation undertaken in this thesis studies effects on cache utilisation, memory saturation, and interconnect utilisation. The performance of the simulated large-scale CMPs and CMCs is attained using real multi-threaded Java applications each of which is run to completion, maintaining complete cache coherence.

The study has shown that CMC architectures provide a feasible approach to the design of future many-core architectures. Multiple CMPs can be replicated across a chip providing another level of abstraction in the design of an architecture. CMC architectures are also able, using task distribution optimisations, to out perform wire delay limited single bus architectures.

8.2 Future Work

The design space for CMP and CMC architectures is vast and the work conducted as part of this thesis has only been able to address a small portion of it, creating many opportunities for future research. The following are some areas where future research projects might be conducted.

Cache and Protocol Optimisations

The multi-level coherence protocol could benefit from several optimisations. Removing the necessity for line inclusion by maintaining a tag–only cache or using Bloom filters [15] for lines that are either stale or shared would reduce the level of redundant lines stored in each cache, potentially improving their efficiency. Support for asymmetric block sizes between levels could better utilise memory bandwidth especially when the number of cores increases, however the impact this would have in an inclusive cache hierarchy is unclear.

In a large multi-level, CMC architecture there is even more potential for exploiting dynamic cache partitioning [128] to utilise each level of shared cache more efficiently. A more thorough investigation into the cost of inclusion through exploration of the cache size, associativity and degree of banking in a coherent environment could lead to a better understanding of the best configurations for constructing multi-level cache hierarchies.

Course grained coherence tracking [112, 25] could be introduced to the system in order to reduce the amount of inclusion within the higher level shared caches. Such a scheme may allow higher level caches to maintain state associated with larger blocks of memory reducing the amount of information stored at each level. The scheme would clearly need to be adaptive to avoid mass invalidations.

Adaptive coherence protocols [35] which attempt to identify migratory data, data which is consistently read and then written, may provide additional benefits in multi-level hierarchies. Data identified as being migratory can be tracked by additional states in the cache and on the initial read the line is serviced in a modified state. This reduces the traffic as the initial downgrade of the line is avoided.

Additionally a more rigorous proof of the coherence protocol and hardware using a formal specification language, such as TLA+[88], would help to test and check the correctness of the system.

Dynamic Exploitation of Locality-Aware Distribution

Utilising locality-aware task distribution within a dynamic execution environment, such as a virtual machine, could provide a more optimal utilisation of the shared caches and interconnect within a CMC, without the need for application restructuring.

The work in this thesis has only evaluated symmetric homogeneous multi-cluster architectures. Prior research has shown the benefit of heterogenous CMPs [83], this work could be extended in the context of CMC architectures by studying both heterogeneous cores and heterogeneous clusters. Dynamic scheduling utilising locality-aware task distribution could be used to distribute simple loop level parallelism to smaller simpler cores under small shared caches, and distribute complex sequential code to more complex cores with larger caches.

Hardware Support for Transactional Memory

Work extending the current simulation models of the architecture to support transactional memory (TM) [65] is currently ongoing. Supporting TM in a multi-cluster hierarchy is an area of research that has not currently been explored, most hardware TM systems extend single bus snooping protocols. However the same limitations imposed by wire-delay within single bus CMP architectures will apply to the scaling of TM architectures relying on single bus snooping protocols.

Bibliography

[1] Jikes^{TM}Research Virtual Machine website, Accessed last 2007. http://jikesrvm.org.

[2] A. Agarwal, D. Chaiken, K. Johnson, D. Kranz, J. Kubiatowicz, K. Kurihara, B.H. Lim, G. Maa, and D. Nussbaum. The MIT Alewife machine: A Large-Scale Distributed-Memory Multiprocessor. In *Workshop on Scalable Shared Memory Multiprocessors*, 1991.

[3] A. Aiken and A. Nicolau. Optimal loop parallelization. In *ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 308–317, 1988.

[4] G. Amdahl. Validity of the single-processor approach to achieving large-scale computer capabilities. In *AFIPS Spring Joint Computer Conference*, volume 30, pages 483–485, 1967.

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

[6] C. Anderson and J.L. Baer. A multi-level hierarchical cache coherence protocol for multiprocessors. In *International Parallel Processing Symposium*, pages 142–148, 1993.

[7] D.W. Anderson, F.J. Sparacio, and R.M. Tomasulo. The IBM System/360 Model 91: Machine Philosophy and Instruction-Handling. *IBM Journal of Research and Development*, 11(1):8–24, 1967.

[8] J. Archibald and J.L. Baer. Cache Coherence Protocols: Evaluation Using a Multiprocessor Simulation Model. *ACM Transactions on Computer Systems*, 4(4):273–298, 1986.

[9] W.C. Athas and C.L. Seitz. Multicomputers: message-passing concurrent computers. *IEEE Computer*, 21(8):9–24, 1988.

[10] T.M Austin and G.S Sohi. Dynamic dependency analysis of ordinary programs. *SIGARCH Computer Architecture News*, 20(2):342–351, 1992.

[11] J.L. Baer and W.H. Wang. On the inclusion properties for multi-level cache hierarchies. In *International Symposium on Computer Architecture*, pages 73–80, 1988.

[12] U.K. Banerjee. *Loop Parallelization*. Kluwer Academic Publishers Norwell, MA, USA, 1994.

[13] L.A. Barroso, K. Gharachorloo, R. McNamara, A. Nowatzyk, S. Qadeer, B. Sano, S. Smith, R. Stets, and B. Verghese. Piranha: a scalable architecture based on single-chip multiprocessing. In *International Symposium on Computer Architecture*, pages 282–293, 2000.

[14] F. Bellard. QEMU, a Fast and Portable Dynamic Translator. In *USENIX Annual Technical Conference, FREENIX Track*, pages 41–46, 2005.

[15] B.H. Bloom. Space/time trade-offs in hash coding with allowable errors. *Communications of the ACM*, 13(7):422–426, 1970.

[16] W. Blume and R. Eigenmann. Performance Analysis of Parallelizing Compilers on the Perfect Benchmarks Programs. *IEEE Transactions on Parallel and Distributed Systems*, 3(6):643–656, 1992.

[17] M.T. Bohr. Interconnect scaling-the real limiter to high performance ULSI. In *International Electron Devices Meeting*, pages 241–244, 1995.

[18] D. Burger and T.M. Austin. The SimpleScalar tool set, version 2.0. *ACM SIGARCH Computer Architecture News*, 25(3):13–25, 1997.

[19] D. Burger and J.R. Goodman. Billion-Transistor Architectures. *IEEE Computer*, 30(9):22–28, 1997.

[20] D. Burger and J.R. Goodman. Billion-Transistor Architectures: There and Back Again. *IEEE Computer*, 37(3):22–28, 2004.

[21] D.R. Butenhof. *Programming with Posix ® Threads*. Addison-Wesley Professional, 1997.

[22] M. Butler, T. Yeh, Y. Patt, M. Alsup, H. Scales, and M. Shebanow. Single instruction stream parallelism is greater than two. In *International Symposium on Computer Architecture*, pages 276–286, 1991.

[23] D. Callahan, K. Kennedy, and K. Porterfield. Software prefetching. In *International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 40–52, 1991.

[24] T.J. Callahan and J. Wawrzynek. Instruction-Level Parallelism for Reconfigurable Computing. In *International Workshop on Field-Programmable Logic and Applications*, pages 248–257, 1998.

[25] J.F. Cantin, J.E. Smith, M.H. Lipasti, A. Moshovos, and B. Falsafi. Coarse-Grain Coherence Tracking: RegionScout and Region Coherence Arrays. *IEEE Micro*, pages 70–79, 2006.

[26] D. Chaiken, C. Fields, K. Kurihara, and A. Agarwal. Directory-based cache coherence in large-scale multiprocessors. *IEEE Computer*, 23(6):49–58, 1990.

[27] S. Chaudhry, P. Caprioli, S. Yip, and M. Tremblay. High-Performance Throughput Computing. *IEEE Micro*, 25(3):32–45, 2005.

[28] T. Chen. An effective programmable prefetch engine for on-chip caches. In *International Symposium on Microarchitecture*, pages 237–242, 1995.

[29] D.R. Cheriton, H.A. Goosen, and P.D. Boyle. Paradigm: a highly scalable shared-memory multicomputer architecture. *IEEE Computer*, 24(2):33–46, 1991.

[30] Y. Chou, B. Fahs, and S. Abraham. Microarchitecture optimizations for exploiting memory-level parallelism. In *International Symposium on Computer Architecture*, pages 76–87, 2004.

[31] F. Chow and J. Hennessy. Register allocation by priority-based coloring. In *SIGPLAN Symposium on Compiler Construction*, pages 222–232, 1984.

[32] L. Codrescu, D.S. Wills, and J. Meindl. Architecture of the Atlas Chip-Multiprocessor: Dynamically Parallelizing Irregular Applications. *IEEE Transactions on Computers*, 50(1):67–82, 2001.

[33] E.G. Coffman, M. Elphick, and A. Shoshani. System Deadlocks. *ACM Computing Surveys*, 3(2):67–78, 1971.

[34] T.P.P. Council. TPC Benchmark C Specification, 2005.

[35] A.L. Cox and R.J. Fowler. Adaptive cache coherency for detecting migratory shared data. In *International Symposium on Computer Architecture*, pages 98–108, 1993.

[36] T. Cramer, R. Friedman, T. Miller, D. Seberger, R. Wilson, and M. Wolczko. Compiling Java just in time. *IEEE Micro*, 17(3):36–43, 1997.

[37] R. Cytron. Doacross: Beyond vectorization for multiprocessors. In *International Conference on Parallel Processing*, pages 836–844, 1986.

[38] V. De and S. Borkar. Technology and design challenges for low power and high performance. In *International Symposium on Low Power Electronics and Design*, pages 163–168, 1999.

[39] K. Diefendorff, P.K. Dubey, R. Hochsprung, and H. Scale. AltiVec extension to PowerPC accelerates media processing. *IEEE Micro*, 20(2):85–95, 2000.

[40] A. Dinn, I. Watson, K. Kirkham, and A. El-Mahdy. The Jamaica Virtual Machine: A Chip Multiprocessor Parallel Execution Environment. Technical report, University of Manchester, 2005.

[41] K. Ebcioglu, E.R. Altman, Y. Heights, and N. York. DAISY: Dynamic Compilation for 100% Architectural Compatibility. In *International Symposium on Computer Architecture*, pages 26–37, 1997.

[42] M. Farrens, G. Tyson, and A.R. Pleszkun. A study of single-chip processor/cache organizations for large numbers of transistors. In *International Symposium on Computer Architecture*, pages 338–347, 1994.

[43] A. Fedorova, M. Seltzer, C. Small, and D. Nussbaum. Performance of multithreaded chip multiprocessors and implications for operating system design. In *USENIX Annual Technical Conference*, pages 26–40, 2005.

[44] M. Fillo, S.W. Keckler, W.J. Dally, N.P. Carter, A. Chang, Y. Gurevich, and W.S. Lee. The M-Machine multicomputer. In *International Symposium on Microarchitecture*, pages 146–156, 1995.

[45] J.A. Fisher. Very Long Instruction Word architectures and the ELI-512. In *International Symposium on Computer Architecture*, pages 140–150, 1983.

[46] K. Flautner, R. Uhlig, S. Reinhardt, and T. Mudge. Thread-level parallelism and interactive performance of desktop applications. *ACM SIGPLAN Notices*, 35(11):129–138, 2000.

[47] K. Flautner, R. Uhlig, S. Reinhardt, and T. Mudge. Thread-level parallelism of desktop applications. In *Workshop on Multi-threaded Execution, Architecture and Compilation*, 2000.

[48] M.J. Flynn. Very high-speed computing systems. *Proceedings of the IEEE*, 54(12):1901–1909, 1966.

[49] S. Frank, H. Burkhardt III, and J. Rothnie. The KSR 1: bridging the gap between shared memory and MPPs. In *IEEE Computer Society International Conference*, pages 285–294, 1993.

[50] C.W. Fraser. A retargetable compiler for ANSI C. *ACM SIGPLAN Notices*, 26(10):29–43, 1991.

[51] M. Galles and E. Williams. Performance optimizations, implementation, and verification of the SGI Challenge multiprocessor. In *Hawaii International Conference on System Sciences*, volume 1, 1994.

[52] R.B Garner, A. Agrawal, F. Briggs, E.W Brown, D. Hough, B. Joy, S. Kleiman, S. Muchnick, M. Namjoo, and D. Patterson. The scalable processor architecture (SPARC). In *IEEE Computer Society International Conference*, pages 278–283, 1988.

[53] P.P. Gelsinger. Microprocessors for the New Millenium: Challenges, Opportunites and New Frontiers. In *IEEE Solid-State Circuits Conference*, pages 22–25, 2001.

[54] J. Goodacre and A.N. Sloss. Parallelism and the ARM instruction set architecture. *IEEE Computer*, 38(7):42–50, 2005.

[55] R. Grindley, T. Abdelrahman, S. Brown, S. Caranci, D. DeVries, B. Gamsa, A. Grbic, M. Gusat, R. Ho, and O. Krieger. The NUMAchine Multiprocessor. Technical Report TR324, University of Toronto, 2000.

[56] M. Gupta and P. Banerjee. Demonstration of Automatic Data Partitioning Techniques for Parallelizing Compilers on Multicomputers. *IEEE Transactions on Parallel and Distributed Systems*, 3(2):179–193, 1992.

[57] M. Gupta and R. Nim. Techniques for Speculative Run-Time Parallelization of Loops. In *IEEE/ACM Conference on Supercomputing*, pages 1–12, 1998.

[58] A. Halaas, B. Svingen, M. Nedland, P. Saetrom, O. Snove Jr., and O.R. Birkeland. A recursive MISD architecture for pattern matching. *IEEE Transactions on Very Large Scale Integration Systems*, 12(7):727–734, 2004.

[59] M.W. Hall, S.P. Amarasinghe, B.R. Murphy, S. Liao, and M.S. Lam. Detecting coarse-grain parallelism using an interprocedural parallelizing compiler. In *Proceedings of Supercomputing*, 1995.

[60] L. Hammond, B.A. Hubbert, M. Siu, M.K. Prabhu, M. Chen, and K. Olukotun. The Stanford Hydra CMP. *IEEE Micro*, 20(2):71–84, 2000.

[61] L. Hammond, K. Olukotun, V. Wong, M. Chen, B.D. Carlstrom, J.D. Davis, B. Hertzberg, M.K. Prabhu, H. Wijaya, and C. Kozyrakis. Transactional Memory Coherence and Consistency. *ACM SIGARCH Computer Architecture News*, 32(2):102, 2004.

[62] J.L. Hennessy and D.A. Patterson. *Computer Architecture: A Quantitative Approach (Third Edition)*. Morgan Kaufmann, 2003.

[63] J.L. Hennessy and D.A. Patterson. *Computer Architecture: A Quantitative Approach (Fourth Edition)*. Morgan Kaufmann, 2006.

[64] M. Herlihy. A methodology for implementing highly concurrent data objects. *ACM Transactions on Programming Languages and Systems*, 15(5):745–770, 1993.

[65] M. Herlihy, J. Eliot, and B. Moss. Transactional Memory: Architectural Support For Lock-free Data Structures. In *International Symposium on Computer Architecture*, pages 289–300, 1993.

[66] R. Ho, K.W. Mai, and M.A. Horowitz. The future of wires. *Proceedings of the IEEE*, 89(4):490–504, 2001.

[67] M. Horsnell. Cycle-Accurate, Distributed Chip Multiprocessor Simulation. In *EPSRC Postgraduate Research in Engineering and Physical Sciences (PREP).*, 2004.

[68] M. Horsnell. Harnessing Java for Novel Chip Multiprocessor Architecture Simulations. In *EPSRC Postgraduate Research in Engineering and Physical Sciences (PREP).*, 2005.

[69] L. Hsu, R. Iyer, S. Makineni, S. Reinhardt, and D. Newell. Exploring the cache design space for large scale CMPs. *ACM SIGARCH Computer Architecture News*, 33(4):24–33, 2005.

[70] W.M.W. Hwu, S.A. Mahlke, W.Y. Chen, P.P. Chang, N.J. Warter, R.A. Bringmann, R.G. Ouellette, R.E. Hank, T. Kiyohara, G.E. Haab, J.G Holm, and D.M Lavery. The superblock: An effective technique for VLIW and superscalar compilation. *Journal of Supercomputing*, 7(1):229–248, 1993.

[71] Intel. Block-Matching In Motion Estimation Algorithms Using Streaming SIMD Extensions 3. Intel Application Note, December 2003. `www.intel.com/cd/ids/developer/asmo-na/eng/dc/pentium4/optimization/66775.htm` accessed last 2007.

[72] R. Iyer, M. Bhat, L. Zhao, R. Illikkal, S. Makineni, M. Jones, K. Shiv, and D. Newell. Exploring Small-Scale and Large-Scale CMP Architectures for Commercial Java Servers. In *IEEE International Symposium on Workload Characterization*, pages 191–200, 2006.

[73] R. Iyer, L. Zhao, F. Guo, R. Illikkal, S. Makineni, D. Newell, Y. Solihin, L. Hsu, and S. Reinhardt. QoS policies and architecture for cache/memory in CMP platforms. In *ACM SIGMETRICS International Conference on Measurement and Modeling of Computer Systems*, pages 25–36, 2007.

[74] T.A. Johnson, R. Eigenmann, and T.N. Vijaykumar. Min-cut program decomposition for thread-level speculation. In *ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 59–70, 2004.

[75] N.P. Jouppi. Improving direct-mapped cache performance by the addition of a small fully-associative cache and prefetch buffers. *SIGARCH Computer Architecture News*, 18(3a):364–373, 1990.

[76] N.P. Jouppi. Cache write policies and performance. In *International Symposium on Computer Architecture*, pages 191–201, 1993.

[77] N.P. Jouppi and D.W. Wall. Available instruction-level parallelism for superscalar and superpipelined machines. In *International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 272–282, 1989.

[78] R. Kalla, B. Sinharoy, and J.M. Tendler. IBM Power5 chip: A dual-core multithreaded processor. *IEEE Micro*, 24(2):40–47, 2004.

[79] G. Kane and J. Heinrich. *MIPS RISC architectures*. Prentice-Hall, Inc. Upper Saddle River, NJ, USA, 1992.

[80] C.N. Keltcher, K.J. McGrath, A. Ahmed, P. Conway, and A.M. Devices. The AMD Opteron processor for multiprocessor servers. *IEEE Micro*, 23(2):66–76, 2003.

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

[82] V. Krishnan and J. Torrellas. Hardware and Software Support for Speculative Execution of Sequential Binaries on a Chip-multiprocessor. In *International Conference on Supercomputing*, pages 85–92, 1998.

[83] R. Kumar, D.M. Tullsen, N.P. Jouppi, and P. Ranganathan. Heterogeneous chip multiprocessors. *IEEE Computer*, 38(11):32–38, 2005.

[84] R. Kumar, V. Zyuban, and D.M. Tullsen. Interconnections in Multi-core Architectures: Understanding Mechanisms, Overheads and Scaling. In *International Symposium on Computer Architecture*, pages 408–419, 2005.

[85] S.R. Kunkel, R.J. Eickemeyer, M.H. Lipasti, T.J. Mullins, B. O'Krafka, H. Rosenberg, S.P. VanderWiel, P.L. Vitale, and L.D. Whitley. A performance methodology for commercial servers. *IBM Journal of Research and Development*, 44(6):851–872, 2000.

[86] L. Lamport. Concurrent reading and writing. *Communications of the ACM*, 20(11):806–811, 1977.

[87] L. Lamport. How to make a correct multiprocess program execute correctly on a multiprocessor. *IEEE Transactions on Computers*, 46(7):779–782, 1997.

[88] L. Lamport. Specifying concurrent systems with TLA+. *Calculational System Design*, 1999.

[89] J.R. Larus. Loop-level parallelism in numeric and symbolic programs. *IEEE Transactions on Parallel and Distributed Systems*, 4(7):812–826, 1993.

[90] J. Laudon. Performance/Watt: the new server focus. *ACM SIGARCH Computer Architecture News*, 33(4):5–13, 2005.

[91] J. Laudon, A. Gupta, and M. Horowitz. Architectural and Implementation Tradeoffs in the Design of Multiple-Context Processors. In *International Symposium on Computer Architecture*, pages 435–435, 1992.

[92] J. Laudon, A. Gupta, and M. Horowitz. Interleaving: A multithreading technique targeting multiprocessors and workstations. In *International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 308–318, 1994.

[93] D. Lea. A Java fork/join framework. In *ACM Conference on Java Grande*, pages 36–43, 2000.

[94] D. Lea. *Concurrent Programming in Java: Design Principles and Patterns*. Addison-Wesley, 2000.

[95] J. Lee, S. Moon, and W. Sung. H.264 decoder optimization exploiting SIMD instructions. In *IEEE Asia-Pacific Conference on Circuits and Systems*, pages 1149–1152, 2004.

[96] R.B. Lee. Subword parallelism with MAX-2. *IEEE Micro*, 16(4):51–59, 1996.

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

[98] D.J. Lilja. Exploiting the parallelism available in loops. *IEEE Computer*, 27(2):13–26, 1994.

[99] J.L. Lo and S.J. Eggers. Improving balanced scheduling with compiler optimizations that increase instruction-level parallelism. *ACM SIGPLAN Notices*, 30(6):151–162, 1995.

[100] J.L. Lo, J.S Emer, H.M. Levy, R.L Stamm, D.M Tullsen, and S.J. Eggers. Converting thread-level parallelism to instruction-level parallelism via simultaneous multithreading. *ACM Transactions on Computer Systems.*, 15(3):322–354, 1997.

[101] K.W. Loveless. The Implementation of Flexible Interconnect in the NUMAchine Multiprocessor. Master's thesis, Department of Electrical and Computer Engineering, University of Toronto, 1996.

[102] K. Mai, T. Paaske, N. Jayasena, R. Ho, W.J. Dally, and M. Horowitz. Smart Memories: a modular reconfigurable architecture. In *International Symposium on Computer Architecture*, pages 161–171, 2000.

[103] P. Marcuello, A. González, and J. Tubella. Speculative multithreaded processors. In *International Conference on Supercomputing*, pages 77–84, 1998.

[104] D. Matzke. Will Physical Scalability Sabotage Performance Gains? *IEEE Computer*, 30(9):37–39, 1997.

[105] O.A. McBryan. An overview of message passing environments. *Parallel Computing*, 20:417–443, 1994.

[106] E. McLellan. The Alpha AXP Architecture and 21064 Processor. *IEEE Micro*, 13(3):36–47, 1993.

[107] C. McNairy and R. Bhatia. Montecito: A Dual-Core, Dual-Thread Itanium Processor. *IEEE Micro*, 25(2):10–20, 2005.

[108] A. Mendelson, J. Mandelblat, S. Gochman, A. Shemer, R. Chabukswar, E. Niemeyer, and A. Kumar. CMP Implementation in Systems based on the Intel®Core™Duo Processor. *Intel® Technology Journal*, 10(2):99–108, May 2006.

[109] P. Merlin and P. Schweitzer. Deadlock Avoidance in Store-and-Forward Networks–I: Store-and-Forward Deadlock. *IEEE Transactions on Communications*, 28(3):345–354, 1980.

[110] J.L. Mitchell, W.B. Pennebaker, C.E. Fogg, and D.J. Legall. *MPEG Video Compression Standard*. Chapman & Hall, Ltd. London, UK, 1996.

[111] G.E Moore. Cramming more components onto integrated circuits. *Electronics*, 38(8):114–117, 1965.

[112] A. Moshovos. RegionScout: exploiting coarse grain sharing in snoop-based coherence. In *International Symposium on Computer Architecture*, pages 234–245, 2005.

[113] T. Mowry and A. Gupta. Tolerating latency through software-controlled prefetching in shared-memory multiprocessors. *Journal of Parallel and Distributed Computing*, 12(2):87–106, 1991.

[114] L.W. Nagel and D.O. Pederson. Simulation Program with Integrated Circuit Emphasis. In *Midwest Symposium on Circuit Theory*, volume 23, 1973.

[115] V. Narayanan, V.K. Paruchuri, E. Cartier, B.P. Linder, N. Bojarczuk, S. Guha, S.L. Brown, Y. Wang, M. Copel, and T.C. Chen. Recent advances and current challenges in the search for high mobility band-edge high-k/metal gate stacks. *Microelectronic Engineering*, 84(9-10):1853–1856, 2007.

[116] B.A. Nayfeh, K. Olukotun, and J.P. Singh. The impact of shared-cache clustering in small-scale shared-memorymultiprocessors. In *International Symposium on High-Performance Computer Architecture*, pages 74–84, 1996.

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

[118] S. Oberman, G. Favor, and F. Weber. AMD 3DNow! technology: architecture and implementations. *IEEE Micro*, 19(2):37–48, 1999.

[119] J.T. Oplinger, D.L. Heine, and M.S. Lam. In Search of Speculative Thread-Level Parallelism. In *International Conference on Parallel Architectures and Compilation Techniques*, pages 303–313, 1999.

[120] G. Ottoni, R. Rangan, A. Stoler, M.J. Bridges, and D.I. August. From Sequential Programs to Concurrent Threads. *IEEE Computer Architecture Letters*, 5(1), 2006.

[121] D.A. Padua, D.J. Kuck, and D.H. Lawrie. High-speed multiprocessors and compilation techniques. *IEEE Transactions on Computers*, 29:763–776, 1980.

[122] D.A. Padua and M.J. Wolfe. Advanced compiler optimizations for supercomputers. *Communications of the ACM*, 29(12):1184–1201, 1986.

[123] S. Palacharla, N.P. Jouppi, and J.E. Smith. Complexity-effective superscalar processors. In *International Symposium on Computer Architecture*, pages 206–218, 1997.

[124] Y.N. Patt, W.M. Hwu, and M. Shebanow. HPS, a new microarchitecture: rationale and introduction. In *Annual Workshop on Microprogramming*, pages 103–108, 1985.

[125] D.A. Patterson. Reduced instruction set computers. *Communications of the ACM*, 28(1):8–21, 1985.

[126] A. Peleg and U. Weiser. MMX Technology Extension to the Intel Architecture. *IEEE Micro*, 16(4):42–50, 1996.

[127] D. Pham, S. Asano, M. Bolliger, M.N Day, H.P. Hofstee, C. Johns, J. Kahle, A. Kameyama, J. Keaty, Y. Masubuchi, M. Riley, D. Shippy, D. Stasiak, M. Suzuoki, M. Wang, J. Warnock, S. Weitzel, D. Wendel, T. Yamazaki, and K. Yazawa. The design and implementation of a first-generation CELL processor - a multi-core SoC. In *International Conference on Integrated Circuit Design and Technology*, pages 49–52, May 2005.

[128] M. Planas, F. Cazorla, A. Ramirez, and M. Valero. Explaining Dynamic Cache Partitioning Speed Ups. *IEEE Computer Architecture Letters*, 6(1), 2007.

[129] M.A Postiff, D.A. Greene, G.S Tyson, and T.N. Mudge. The limits of instruction level parallelism in SPEC95 applications. *SIGARCH Computure Architecture News*, 27(1):31–34, 1999.

[130] M.K. Prabhu and K. Olukotun. Using thread-level speculation to simplify manual parallelization. In *International Symposium on Principles and Practice of Parallel Programming*, pages 1–12, 2003.

[131] D.J. Quammen, D.R. Miller, and D. Tabak. Register window management for a real-time multitasking RISC. In *Hawaii International Conference on System Sciences*, volume 1, 1989.

[132] G. Radin. The 801 minicomputer. In *International Symposium on Architectural Support for Programming Languages and Operating Systems*, pages 39–47, 1982.

[133] R. Rajwar and J.R. Goodman. Speculative lock elision: Enabling highly concurrent multithreaded execution. In *International Symposium on Microarchitecture*, pages 01–05, 2001.

[134] R. Rajwar, M. Herlihy, and K. Lai. Virtualizing transactional memory. In *International Symposium on Computer Architecture*, pages 494–505, 2005.

[135] L. Rauchwerger and D.A. Padua. The LRPD test: speculative run-time parallelization of loops with privatization and reduction parallelization. *IEEE Transactions on Parallel and Distributed Systems*, 10(2):160–180, 1999.

[136] J.H. Saltz, R. Mirchandaney, and K. Crowley. Run-time parallelization and scheduling of loops. *IEEE Transactions on Computers*, 40(5):603–612, 1991.

[137] K. Sankaralingam, R. Nagarajan, H. Liu, C. Kim, J. Huh, D. Burger, S.W. Keckler, and C.R. Moore. Exploiting ILP, TLP, and DLP with the polymorphous TRIPS architecture. *SIGARCH Computer Architecture News*, 31(2):422–433, 2003.

[138] T. Scholz and M. Schafers. An Improved Dynamic Register Array Concept for High-Performance RISC Processors. In *Hawaii International Conference on System Sciences*, 1995.

[139] Semiconductor Industry Association. *The International Technology Roadmap for Semiconductors*, 2005.

[140] R.L. Sites. *Alpha Architecture Reference Manual*. Digital Press, 1998.

[141] A. Smith, J. Gibson, B. Maher, N. Nethercote, B. Yoder, D. Burger, K.S McKinle, and J. Burrill. Compiling for EDGE Architectures. In *International Symposium on Code Generation and Optimization*, pages 185–195, 2006.

[142] B.J. Smith. Architecture and applications of the HEP multiprocessor computer system. *Real-time signal processing IV*, pages 241–248, 1982.

[143] J.E. Smith and S. Vajapeyam. Trace processors: moving to fourth-generation microarchitectures. *IEEE Computer*, 30(9):68–74, 1997.

[144] L.A. Smith, J.M. Bull, and J. Obdrzalek. A Parallel Java Grande Benchmark Suite. In *ACM/IEEE Conference on Supercomputing*, 2001.

[145] G.S. Sohi, S.E. Breach, and T.N. Vijaykumar. Multiscalar processors. In *International Symposium on Computer Architecture*, pages 414–425, 1995.

[146] J.G. Steffan, C.B. Colohan, A. Zhai, and T.C. Mowry. A scalable approach to thread-level speculation. In *International Symposium on Computer Architecture*, pages 1–12, 2000.

[147] J.G. Steffan, C.B. Colohan, A. Zhai, and T.C. Mowry. Improving value communication for thread-level speculation. In *International Symposium on High-Performance Computer Architecture*, pages 65–75, 2002.

[148] J.G. Steffan and T. Mowry. The Potential for Using Thread-Level Data Speculation to Facilitate Automatic Parallelization. In *International Symposium on High-Performance Computer Architecture*, volume 15, 1998.

[149] P. Stenström. A survey of cache coherence schemes for multiprocessors. *IEEE Computer*, 23(6):12–24, 1990.

[150] P. Stenström, E. Hagersten, D.J. Lilja, M. Martonosi, and M. Venugopal. Trends in shared memory multiprocessing. *IEEE Computer*, 30(12):44–50, 1997.

[151] J. Subhlok and G. Vondran. Optimal latency-throughput tradeoffs for data parallel pipelines. In *ACM Symposium on Parallel Algorithms and Architectures*, pages 62–71, 1996.

[152] Sun Microsystems, Inc., 4150 Network Circle, Santa Clara, California 95054, U.S.A. *OpenSPARC^{TM} T1 Microarchitecture Specification*, Revision A edition, August 2006.

[153] P. Sweazey and A.J. Smith. A class of compatible cache consistency protocols and their support by the IEEE futurebus. In *International Symposium on Computer Architecture*, pages 414–423, 1986.

[154] M. Takahashi, H. Takano, E. Kaneko, and S. Suzuki. A shared-bus control mechanism and a cache coherence protocol for ahigh-performance on-chip multiprocessor. In *International Symposium on High-Performance Computer Architecture*, pages 314–322, 1996.

[155] M.R. Thistle and B.J. Smith. *A processor architecture for horizon.* IEEE Computer Society Press Los Alamitos, CA, USA, 1988.

[156] R.M. Tomasulo. An Efficient Algorithm for Exploiting Multiple Arithmetic Units. *IBM Journal of Research and Development*, 11(1):25–33, 1967.

[157] M. Tremblay, J.M. Narayanan, and V.L. He. VIS speeds new media processing. *IEEE Micro*, 16(4):10–20, 1996.

[158] M. Tremblay and J.M. O'Connor. UltraSparc I: a four-issue processor supporting multimedia. *IEEE Micro*, 16(2):42–50, 1996.

[159] D.M. Tullsen, S.J. Eggers, and H.M. Levy. Simultaneous multithreading: maximizing on-chip parallelism. In *International Symposium on Computer Architecture*, pages 392–403, 1995.

[160] D.M. Ungar. *The design and evaluation of a high performance Smalltalk system.* MIT Press Cambridge, MA, USA, 1987.

[161] T. Ungerer, B. Robič, and J. Šilc. A survey of processors with explicit multithreading. *ACM Computing Surveys*, 35(1):29–63, 2003.

[162] S.P. VanderWiel and D.J Lilja. Data prefetch mechanisms. *ACM Computing Surveys*, 32(2):174–199, 2000.

[163] S.P. VanderWiel, D. Nathanson, and D.J. Lilja. Complexity and performance in parallel programming languages. In *International Workshop on High-Level Programming Models and Supportive Environments*, pages 3–12, 1997.

[164] D. Vuyst, R. Kumar, and D.M. Tullsen. Exploiting unbalanced thread scheduling for energy and performance on a CMP of SMT processors. In *Parallel and Distributed Processing Symposium*, page 10, 2006.

[165] E. Waingold, M. Taylor, D. Srikrishna, V. Sarkar, W. Lee, V. Lee, J. Kim, M. Frank, P. Finch, R. Barua, J. Babb, S. Amarasinghe, and A. Agarwal. Baring It All to Software: Raw Machines. *IEEE Computer*, 30(9):86–93, 1997.

[166] D.W. Wall. Limits of Instruction Level Parallelism. Technical report, Digital - Western Research Laboratory, 1993.

[167] D.W. Wall. Speculative execution and instruction-level parallelism. Technical report, Digital - Western Research Laboratory, 1994.

[168] A.W. Wilson Jr. Hierarchical cache/bus architecture for shared memory multiprocessors. In *International Symposium on Computer Architecture*, pages 244–252, 1987.

[169] S.C. Woo, M. Ohara, E. Torrie, J.P. Singh, and A. Gupta. The SPLASH-2 programs: characterization and methodological considerations. In *International Symposium on Computer Architecture*, pages 24–36, 1995.

[170] Greg M. Wright. *A single chip multi-processor architecture with hardware thread support.* PhD thesis, School of Computer Science, University of Manchester, 2001.

[171] T.Y. Yeh and Y.N. Patt. Two-level adaptive branch prediction. In *International Symposium and Workshop on Microarchitecture*, 1991.

[172] J. Zhao, I. Rogers, C. Kirkham, and I. Watson. Loop Parallelisation for the Jikes RVM. In *International Conference on Parallel and Distributed Computing, Applications and Technologies*, pages 35–39, 2005.

[173] C. Zilles and G. Sohi. Execution-based prediction using speculative slices. In *International Symposium on Computer Architecture*, pages 2–13, 2001.

APPENDIX A

Jamaica - Instruction Set Architecture

The Jamaica instruction set borrows some of its instruction formats from the Digital Alpha 32-bit instruction set architecture [140], but is not binary compatible.

A.1 Instruction Formats

The architecture supports four distinct instruction formats, register form (Figure A.1), immediate form (Figure A.2), branch form (Figure A.3) and memory form (Figure A.4).

A.1.1 Register Form

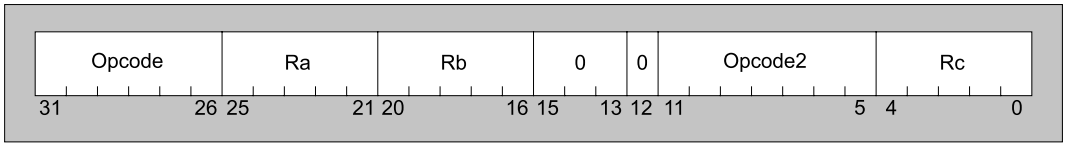


Figure A.1: ***Register form*** $R_c \leftarrow R_a \ op \ R_b$.

A.1.2 Immediate Form

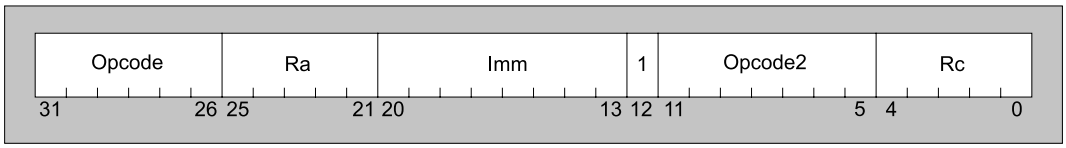


Figure A.2: ***Register immediate form*** $R_c \leftarrow R_a \ op \ R_b$.

A.1.3 Branch Form

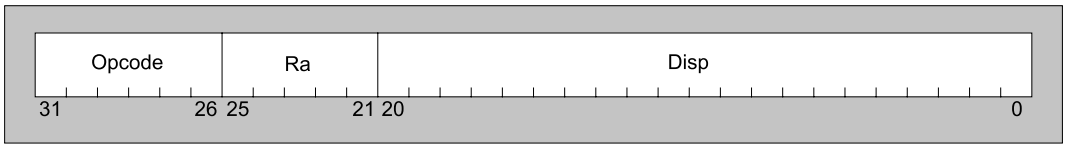


Figure A.3: ***Branch form.***

A.1.4 Memory Form

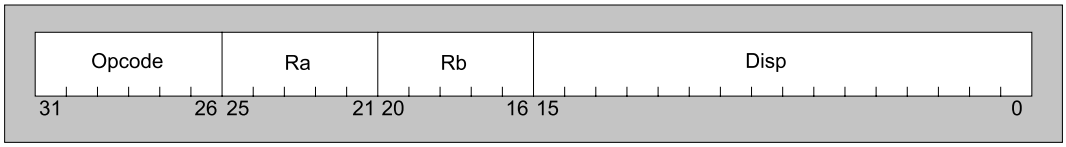


Figure A.4: ***Memory form.***

A.2 Instruction Set

A.2.1 Arithmetic/Logical Instructions

Register Form: `OP` R_a, R_b, R_c

Register Immediate Form: R_a, `Imm`, R_c

ADD	$R_c \leftarrow R_a + R_b$
SUB	$R_c \leftarrow R_a - R_b$
CMPEQ	$R_c \leftarrow (R_a = R_b)$
CMPLE	$R_c \leftarrow (R_a <= R_b)$
CMPLT	$R_c \leftarrow (R_a < R_b)$
CMPULE	$R_c \leftarrow (R_a <= R_b)$
CMPULT	$R_c \leftarrow (R_a < R_b)$
S4ADD	$R_c \leftarrow (4 \times R_a + R_b)$
S8ADD	$R_c \leftarrow (8 \times R_a + R_b)$
S4SUB	$R_c \leftarrow (4 \times R_a - R_b)$
S8SUB	$R_c \leftarrow (8 \times R_a - R_b)$
AND	$R_c \leftarrow (R_a \,\&\, R_b)$
BIC	$R_c \leftarrow (R_a \,\&\, \backsim R_b)$
BIS	$R_c \leftarrow (R_a \mid R_b)$
EQV	$R_c \leftarrow (R_a \mid \backsim R_b)$
ORNOT	$R_c \leftarrow (R_a \mid \backsim R_b)$
XOR	$R_c \leftarrow (R_a \wedge R_b)$
SLL	$R_c \leftarrow (R_a \ll R_b)$
SRL	$R_c \leftarrow (R_a \lll R_b)$
SRA	$R_c \leftarrow (R_a \gg R_b)$
CMOVEQ	if$(R_a = 0)$ $R_c \leftarrow R_b$
CMOVGE	if$(R_a \geqslant 0)$ $R_c \leftarrow R_b$
CMOVGT	if$(R_a > 0)$ $R_c \leftarrow R_b$
CMOVLBC	if$((R_a \wedge 1) = 0)$ $R_c \leftarrow R_b$
CMOVLBS	if$((R_a \wedge 0) = 0)$ $R_c \leftarrow R_b$
CMOVLE	if$(R_a \leq 0)$ $R_c \leftarrow R_b$
CMOVLT	if$(R_a < 0)$ $R_c \leftarrow R_b$
CMOVNE	if$(R_a \neq 0)$ $R_c \leftarrow R_b$
MUL	$R_c \leftarrow R_a \times R_b$
TRQ	see Section 6.2.2
RCR	$R_c \leftarrow$ CReg$[R_b]$
WCR	CReg$[R_b] \leftarrow R_a$
CAS	single word compare and swap
SIRQ	Send IRQ R_b to thread with ID R_a
EVICT	Evict a frame; $R_c \leftarrow$ 1 or 0

Table A.1: *Jamaica instruction set: arithmetic/logical instructions.*

A.2.2 Control Transfer Instructions

Branch Form: `OP` R_a`, disp` 21-bit signed displacement

BEQ	Branch if $R_a = 0$
BGE	Branch if $R_a \geq 0$
BGT	Branch if $R_a > 0$
BLBC	Branch if $R_a \& 1 = 0$
BLBS	Branch if $R_a \& 0 = 0$
BLE	Branch if $R_a \leq 0$
BLT	Branch if $R_a < 0$
BNE	Branch if $R_a \neq 0$
BR	Branch
BSR	Branch to subroutine
THB	Thread branch (following TRQ)

Table A.2: *Jamaica instruction set: branch form control instructions.*

Memory Form: `OP` R_a`, disp` 16-bit signed displacement

JSR	Jump to subroutine
JMP	Jump
RET	Return (takes address from `%i7`)
THJ	Thread jump (following TRQ)
RTI	Return from interrupt

Table A.3: *Jamaica instruction set: memory form control instructions.*

A.2.3 Memory Instructions

Memory Form: `OP` R_a`, disp,` R_b 16-bit signed displacement

LDA	$R_a \leftarrow$ disp + R_b
LDAH	$R_a \leftarrow$ disp $\ll$ 16 + R_b
LDL	$R_a \leftarrow$ Mem[disp + R_b]
STL	Mem[disp + R_b] $\leftarrow$ R_a
LDB	$R_a \leftarrow$ Mem[disp + R_b], byte, sign-extended
LDBU	$R_a \leftarrow$ Mem[disp + R_b], byte, zero-extended
STB	Mem[disp + R_b] $\leftarrow$ R_a, byte
LDL_L	$R_a \leftarrow$ Mem[disp + R_b], set lock_base, set lock_flag
STL_C	if(lock_flag) {Mem[disp + R_b] $\leftarrow$ R_a; $R_a \leftarrow$ 1} else { $R_a \leftarrow$ 0}
WAIT	Sleep until lock_flag is cleared

Table A.4: *Jamaica instruction set: memory instructions.*

A.3 BuiltIn Instructions

Trap address	BuiltIn	Description
0xffff00d0	contextReplace	switch context registers with memory
0xffff00d4	printTimeStamp	prints processor and context id with cycle count
0xffff00d8	getCycleCount	returns cycle count in register %o0
0xffff0104	fstat	unix fstat equivalent
0xffff0144	copyMemory	copy block of memory at %o0 to %o1, %o2 bytes
0xffff0148	setMemory	set block of memory at %o0 to value %o1, %o2 bytes
0xffff0160	zeroCtxStats	zero statistics for a given processor context
0xffff0164	reportCtxStats	report statistics for a given processor context
0xffff012c	fflush	unix fflush equivalent
0xffff0000	simExit	forceably quit the sim
0xffff0008	fopen	unix fopen equivalent
0xffff000c	fputc	unix fputc equivalent
0xffff0010	fgetc	unix fgetc equivalent
0xffff0018	ungetc	unix ungetc equivalent
0xffff00e0	zeroPerf	zero all statistic counters
0xffff00ec	getNumProcs	return total number of processors (*deprecated*)
0xffff00f0	getNumCtxs	returns number of contexts
0xffff0108	open	unix open equivalent
0xffff0124	lseek	unix lseek equivalent
0xffff011c	read	unix read equivalent
0xffff014c	reportPerf	report all statistic counters
0xffff0120	write	unix write equivalent
0xffff0128	getTimeOfDay	get unix time
0xffff010c	close	unix close equivalent
0xffff0138	notifyDebugger	notify the debugger of object updates in VM
0xffff0200	switchCaches	switch to cycle-level cache models
0xffff0300	utilityCall	2 integer inputs, 1 integer output utility call

Table A.5: *Jamaica instruction set: builtin instructions.*