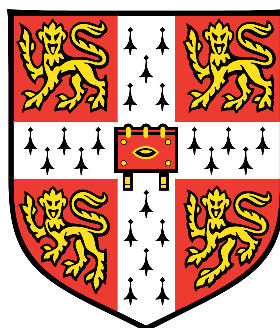
Proximity Coherence for Chip-Multiprocessors



Nick Barrow-Williams

Trinity Hall

University of Cambridge

This dissertation is submitted for the degree of

Doctor of Philosophy

January 2011

Abstract

Many-core architectures provide an efficient way of harnessing the growing numbers of transistors available in modern fabrication processes; however, the parallel programs run on these platforms are increasingly limited by the energy and latency costs of communication. Existing designs provide a functional communication layer but do not necessarily implement the most efficient solution for chip-multiprocessors, placing limits on the performance of these complex systems. In an era of increasingly power limited silicon design, efficiency is now a primary concern that motivates designers to look again at the challenge of cache coherence.

The first step in the design process is to analyse the communication behaviour of parallel benchmark suites such as Parsec and SPLASH-2. This thesis presents work detailing the sharing patterns observed when running the full benchmarks on a simulated 32-core x86 machine. The results reveal considerable locality of shared data accesses between threads with consecutive operating system assigned thread IDs. This pattern, although of little consequence in a multi-node system, corresponds to strong physical locality of shared data between adjacent cores on a chip-multiprocessor platform.

Traditional cache coherence protocols, although often used in chip-multiprocessor designs, have been developed in the context of older multi-node systems. By redesigning coherence protocols to exploit new patterns such as the physical locality of shared data, improving the efficiency of communication, specifically in chip-multiprocessors, is possible. This thesis explores such a design – Proximity Coherence – a novel scheme in which L1 load misses are optimistically forwarded to nearby caches via new dedicated links rather than always being indirected via a directory structure.

Declaration

This dissertation is my own work and contains nothing which is the outcome of work done in collaboration with others, except as specified in the text. Including tables and footnotes, this dissertation does not exceed 60,000 words.

Acknowledgements

First and foremost I would like to thank my supervisor Simon Moore for his guidance. Additionally, I would like to acknowledge Robert Mullins' invaluable role during my studies. His passion for research, and patient support proved essential.

I would like to thank all the students of the Cambridge computer architecture research group, and Trinity Hall, for the many stimulating discussions and enjoyable friendships over the past three years.

A special mention must go to Chris Fensch. Without his support I would not be where I am today. His belief in my abilities and dedication to my progress allowed me to get through the hard times and for this I am immensely grateful.

Finally, I wish to thank my family for their unwavering support throughout my time in Cambridge.

Contents

1 Introduction **1**

1.1 Thesis . . . 3

1.2 Contributions . . . 3

1.3 Publications . . . 4

1.4 Thesis Outline . . . 4

2 Background **6**

2.1 Chip-Multiprocessor Architectures . . . 6

2.1.1 Architectural Characteristics . . . 7

2.1.2 Research Architectures . . . 8

2.1.2.1 MIT Raw . . . 8

2.1.2.2 Intel Polaris . . . 11

2.1.2.3 Intel Larrabee . . . 12

2.1.3 Industry Architectures . . . 14

2.1.3.1 Tilera Tile64 . . . 15

2.1.3.2 Intel Core 2 . . . 17

2.1.3.3 Sun UltraSPARC T2 . . . 18

2.1.3.4 IBM Cell . . . 20

2.2 Interconnection Networks . . . 23

2.2.1 Bus-based Interconnects . . . 23

2.2.2 Network-on-Chip . . . 24

2.2.3 Flow-control Mechanisms . . . 25

2.2.4 Low-Latency Router Design 27
2.3 Cache Coherence 28
2.3.1 Consistency 28
2.3.2 Coherence 30
2.3.3 Memory Models 31
2.3.4 Snooping and Directory Systems 32
2.3.5 Distributed Systems 34
2.3.6 Protocols 35
2.3.7 Summary 37
2.4 Parallel Benchmarks 38
2.4.1 Shared Memory Synchronisation Methods 39
2.4.2 SPLASH-2 40
2.4.3 Parsec 42
2.5 Summary 43
2.5.1 Processor Architecture 43
2.5.2 Cache Coherence 44
2.5.3 Parallel Benchmarks 44
2.5.4 Interconnection Networks 45

3 Communication Characterisation **46**
3.1 Introduction 46
3.2 Benchmark Background 47
3.3 Sharing Pattern Background 49
3.4 Evaluation Setup 52
3.5 Experimental Results 54
3.5.1 Communicating Accesses 55
3.5.2 Communication Patterns 58
3.5.3 Sharing Patterns 68
3.5.4 Read-Set Stability 74
3.6 Conclusions 77

4 Physical Locality 79

4.1 Initial Study 80

5 Proximity Coherence 88

5.1 Introduction 88

5.2 Motivation 90

5.2.1 Proximity Hits 90

5.2.2 Baseline Architecture 90

5.2.3 Concurrent Proximity Requests 92

5.2.4 Energy Considerations 92

5.2.5 Summary 93

5.3 Proximity Coherence 93

5.3.1 An Example of Proximity Coherence 95

5.3.2 Invalidations 96

5.3.3 L1 Cache Replacements 99

5.3.4 Forwarding from Modified and Exclusive 103

5.3.5 State Machine Description 104

5.3.5.1 L1 States 104

5.3.5.2 L2/Directory States 107

5.3.6 Race Conditions 107

5.3.6.1 PROXREQ against INVALIDATE 109

5.3.6.2 UPDATE_S against INVALIDATE 109

5.3.6.3 Write-backs from State **F** 110

5.3.6.4 UPGRADE of a line in **F** 111

5.3.7 Hardware Costs 112

6 System Evaluation 113

6.1 Introduction 113

6.2 Evaluation Setup 113

6.2.1 Simulation Parameters 114

6.2.2 Benchmark Selection 115

6.2.3 Thread Mapping 116
6.3 Experimental Results 117
6.3.1 Impact on Memory Latency 117
6.3.2 Invalidation Chain Length 120
6.3.3 Proximity Hit Rate 120
6.3.4 Execution Time Improvements 121
6.3.5 Impact on Network Traffic 122
6.3.6 Impact on Energy 125
6.4 Conclusion 126

7 Related Work **127**
7.1 Communication Characterisation 127
7.2 Proximity Coherence 128

8 Conclusions **134**
8.1 Thesis Summary 134
8.2 Future Directions 136
8.2.1 Communication Characterisation 137
8.2.2 Proximity Coherence 137

Bibliography **149**

List of Figures

2.1 The Raw architecture, reproduced from original publication [73]. 9
2.2 The Intel Polaris architecture, reproduced from Intel materials. 11
2.3 Intel Larrabee. 13
2.4 The Tilera Tile64 architecture, reproduced from original publication [11]. 15
2.5 Intel Core 2 "Conroe" architecture. 17
2.6 The Sun UltraSPARC T2 architecture, reproduced from original publication [55]. . 19
2.7 The IBM Cell BE architecture, reproduced from original publication [47]. 21
2.8 A generic bus architecture. 24
2.9 A generic network-on-chip architecture. 25
2.10 A basic MESI protocol state diagram. 36

3.1 Communicating and non-communicating memory accesses. 48
3.2 A read-only sharing memory access pattern. 50
3.3 A migratory sharing memory access pattern. 50
3.4 A producer-consumer memory access pattern. 51
3.5 Fraction of read and write accesses to shared memory locations that communicate data. A read is considered communicating when it reads a value that has been produced by another processor and has not been read before. A write is considered communicating when it produces a value that is read by a different processor. 56
3.6 Instructions per communicating read and write accesses. 57

3.7 Communication between different cores during the entire parallel phase of the program for the SPLASH-2 benchmark suite, normalised per application. 59
3.8 Communication between different cores during the entire parallel phase of the program for the Parsec benchmark suite, normalised per application. 60
3.9 Communication between private caches of a 32 processor system running the SPLASH-2 benchmarks, normalised per application. 62
3.10 Communication between private caches of a 32 processor system running the Parsec benchmarks, normalised per application. 63
3.11 Communication between different threads during the entire parallel phase of the benchmark, normalised per application. 65
3.12 Communication changes over time for a selection of processors and applications, normalised per application. 67
3.13 Analysis of the read-only sharing pattern. The spatial analysis shows the percentage of the shared address space that is used according to the read-only sharing pattern. The quantitative analysis shows the percentage of reads to shared address space that access a location that had been classified as read-only. For both analyses, the number of processors the line is read by is used to classify each access (Read-only locations with only one reading processor, are written by a different processor). 69
3.14 Analysis of the migratory sharing pattern. The spatial analysis shows the percentage of the shared address space that is used according to the migratory sharing pattern. The quantitative analysis shows the percentage of communicating writes that access a location that had been classified as migratory. For both analyses, the number of processors participating in the migratory patter is used to classify the write accesses. 70

3.15 Analysis of the producer-consumer sharing pattern. The spatial analysis shows the percentage of the shared address space that is used according to the producer-consumer sharing pattern. The quantitative analysis shows the percentage of communicating writes that access a location that had been classified as producer consumer. In both experiments, accesses are classified by how many processors consume the data. 71
3.16 Stability analysis of the read set of produced values. In order to characterise the stability of a location, it is necessary that at least two communicating writes are performed. The spatial analysis shows the percentage of shared address space with two or more communicating writes. The quantitative analysis shows the percentage of communicating writes that access a location with two or more communicating writes. For both analyses, accesses are classified according to the read set stability for the relevant memory location. 75

4.1 Results of the oracle study to investigate the limits of physical locality in a 32 core system. When an L1 cache miss occurs, the core checks all L1s for data that could be forwarded. 81
4.2 Impact on the proximity hit rate of SPLASH-2 benchmarks when the number of cores snooped is reduced. In this study, snooping is limited to n neighbours. The study evaluates three policies for neighbour selection: *Ideal*, *Random* and *H-tree* . 82
4.3 Impact on the proximity hit rate of Parsec benchmarks when the number of cores snooped is reduced. In this study, snooping is limited to n neighbours. The study evaluates three policies for neighbour selection: *Ideal*, *Random* and *H-tree* 83
4.4 An example random thread mapping used to evaluate physical locality 85
4.5 An H-tree thread mapping used to evaluate physical locality 86

5.1 Top left corner of a tiled many-core processor. Grey connections show the global on-chip interconnect. Black connections show the proximity links between the L1D caches. 91
5.2 Example of a *proximity hit*. 94

5.3 Example of a *proximity miss*. P_A misses in its local cache and sends out four PROXREQ messages to its neighbouring cores (step ❶). Since none of these tiles can provide the data, they all respond with a PROXMISS message each. This situation is called a *proximity miss* (step ❷). P_A now sends a GETS message to the directory in order to request the data (step ❸). 95
5.4 Example of external invalidations. For this cache line, the directory is located in core P27, indicated by solid grey shading. 97
5.5 Example of internal invalidations. For this cache line, the directory is located in core P27, indicated by solid grey shading. 100
5.6 Example of an L1 replacement in case of forwarded data. For this cache line, the directory is located in core P26, indicated by solid gray shading. 101
5.7 A diagram of the stable L1 states in the Proximity Coherence. Standard MESI transitions are greyed out for clarity. 105

6.1 Thread mapping considerations: (a) shows the best neighbour lists for *fmm* and *water-nsquared*. A darker colour indicates that this core is more likely to be able to forward data to the requesting core. There is a dark region around the diagonal, which resulted in the approximate thread placement strategy shown in (b). 115
6.2 Cache miss latency reduction in % compared to a system using the MESI baseline protocol. 118
6.3 Distribution of the depth of the sharer graph at the time of an invalidation request, when using the *ProxF* scheme. The graph has in most cases only a depth of 1, resulting in negligible overhead. The vertical dashed line indicates the maximum depth observed in that program. 119
6.4 Measured proximity hit rates for *Prox* and *ProxF*. 121
6.5 Runtime reduction compared to a system using the MESI baseline protocol. For increased clarity, the y-axis is scaled to show runtimes between 0.85x and 1.10x . 122
6.6 Normalised network traffic compared to a system using the MESI baseline protocol. “B” refers to the baseline system, “P” refers to *Prox*, “F” refers to *ProxF*, and “N” refers to *ProxF-N*. 123

6.7 Normalised estimated network energy consumption compared to a system using the MESI baseline protocol. “B” refers to the baseline system, “P” refers to *Prox*, “F” refers to *ProxF*, and “N” referes to *ProxF-N*. Additionally shown is the energy required to perform a cache lookup in the case of a servicing a proximity request 124

List of Tables

2.1 A breakdown of instructions executed in a 32 processor machine running SPLASH-2, reproduced from original publication [79]. 41
2.2 A qualitative description of the benchmarks in the Parsec suite, reproduced from original publication [15]. 43

3.1 Simulated system parameters 53
3.2 The minimum, maximum and average number of communicating writes per line, i.e. those shown in Figure 3.16. 76

4.1 The hit rates of Random and H-Tree mappings normalised to that of the Ideal mapping. Results are shown for a snoop width of 4 87

5.1 The stable states implemented in the GEMS model of each L1 cache. 105
5.2 The transient states implemented in the GEMS model of each L1 cache. 106
5.3 The stable states implemented in the GEMS model of each L2 cache. 107
5.4 The transient states implemented in the GEMS model of each L2 cache. 107

6.1 Parameters used in the full-system simulation to evaluate Proximity Coherence. . . 114

CHAPTER 1

Introduction

The past decade has seen a dramatic shift towards multicore designs as the dominant processor architecture. Although the number of transistors available to designers is still rising in accordance with Moore's Law [59], it has now become impossible to use these extra resources to augment existing single-core designs [2; 63; 64; 76]. In particular, the power consumption and latency penalty of on-chip wiring has limited the feasible size and complexity of a single core [28]. Such issues are commonly referred to as the 'power wall', and dealing with this problem, while still improving performance, has become the primary concern of processor designers. A widely adopted solution is for processor architects to move away from single-core designs towards multicore platforms. Multicore designs use the growing number of transistors to add additional compute cores to the die, foregoing any significant increase in performance of each individual core. This paradigm shift introduces many new challenges for computer architects, who must now design systems to exploit thread-level parallelism in order to increase system performance. Furthermore, designers must strive for all aspects of efficiency in all parts of the design – in particular, performance per Watt.

Producing a truly efficient system requires a deep understanding of the costs of each aspect of a processor architecture. Additional challenges lie in optimising for modern process technologies that exhibit drastically different physical characteristics from even those used only a matter of years ago. In particular, the costs of communication relative to computation have grown considerably [60]. However, although there are challenges,

there are also a great many opportunities. Designers must now readdress all aspects of existing designs to evaluate potential tradeoffs between communication and computation.

It is first necessary to establish the meaning of communication in the context of this work. In typical chip-multiprocessors, the predominant communication mechanism is the memory subsystem, and this is especially true in traditional shared memory architectures. Such architectures use cache coherence to provide communication between parallel processing cores – hence this is the most logical system to examine for potential efficiency improvements.

When re-evaluating previous design decisions such as cache coherence, architects must now employ workload driven analysis of any proposed changes. Specialisation leads to efficiency, and even in the case of general-purpose architectures, it is now essential to only spend power in the most valuable parts of the design, as determined by the applications to be run. To produce the most efficient hardware designs, it is now vital to have a detailed understanding of application behaviour, as without this knowledge it is extremely difficult to correctly partition resources between communication and computation. This is particularly true for recent benchmark suite releases such as Parsec that specifically utilise the tightly coupled cores available in chip-multiprocessors to allow the use of newer, high performance, models of parallelisation. These software techniques introduce additional irregularity and complexity to data sharing and are entirely dependent on efficient communication between processors to provide good scalability.

This thesis shows that with careful analysis of the communication patterns observed in chip-multiprocessors, it is possible to design efficient coherence protocols suitable for modern multicore architectures. In particular this work details how the exploitation of the physical layout of the cache hierarchy, combined with the specialisation of portions

of the interconnection network, can both increase performance, while reducing energy consumption.

1.1 Thesis

Analysis of communication patterns in shared-memory parallel applications facilitates the design of a locality-aware cache coherence protocol for chip-multiprocessors.

1.2 Contributions

In conducting my research, I have made the following contributions to the field:

- Comprehensive analysis of communication patterns in both legacy and emerging shared-memory applications.
- Discovery of physical locality – shared data is often found in nearby caches – in many parallel benchmark applications.
- Proposal of low-cost links between physically local tiles to be used specifically to exploit this new locality.
- Design and evaluation of Proximity Coherence, a new protocol to use the low-cost local links to improve performance and reduce energy consumption of shared-memory chip-multiprocessors.

1.3 Publications

The communication characterisation work in Chapter 3 has been published and presented at the IEEE International Symposium on Workload Characterisation (IISWC) [9]. The design and evaluation of Proximity Coherence, covered in Chapters 5 and 6 has been published and presented at the ACM/IEEE conference on Parallel Architectures and Compilation Techniques (PACT) [10].

1.4 Thesis Outline

Chapter 2 presents the background to the baseline chip-multiprocessor architecture used throughout this work. The areas of tiled architectures, cache hierarchies, coherence, network-on-chip and parallel benchmarks are covered, at each stage explaining how the baseline architecture is derived.

Chapter 3 details the experimental method and results of a communication characterisation of two popular parallel benchmark suites, SPLASH-2 [79] and PARSEC [14]. The spatial and temporal patterns in the communicating memory accesses of each program are described, as well as a classification of the accesses to regions of shared memory.

Chapter 4 describes the impact of the new "physical locality" found in the characterisation work, and in particular, its importance to memory system design and relationship to thread mapping.

Chapter 5 covers the design and mechanism of a novel cache coherence protocol - Proximity Coherence. The full state machine is presented, and details of race condition mechanisms are described.

Chapter 6 contains the evaluation of Proximity Coherence. The impact of thread

mapping is considered, followed by analysis of the impact on latency, network traffic, and energy consumption.

Chapter 7 describes similarities to and differences from related works in the fields of workload characterisation and coherence protocol design.

Chapter 8 concludes the thesis, summarising how the work deals with the difficult challenge of coherence in chip-multiprocessors. Finally, further directions of research are discussed, including several possible extensions to enhance Proximity Coherence.

CHAPTER 2

Background

Processor design has undergone a fundamental transition in recent years. Multiprocessor systems, and in particular highly integrated multicore designs, are now the standard computing platform, with many such offerings emerging from both academia and industry. The work in this thesis investigates the challenges faced when increasing the core count of these new systems. However, to appreciate the challenges posed by continued scaling of multiprocessor designs, it is necessary to first analyse the design processes leading to the current generation of systems.

This background chapter introduces the concepts fundamental to the design of chip-multiprocessors; tiled architectures, cache hierarchies, coherence and consistency, network-on-chip and parallel benchmarks. The design choices presented by each topic are discussed with particular focus on those parameters chosen for the baseline system used in the analysis found in Chapters 3, 4, 5 and 6. Prior work related to the new research presented in this thesis can be found in Chapter 7.

2.1 Chip-Multiprocessor Architectures

The continuing growth in the number of transistors available to hardware designers has long been the driving force behind the rapid improvements of computing power in modern architectures. For many years these transistors have been used to increase the depth of instruction-level parallelism that can exploited by a single processor. This has largely

involved increasing the complexity of the many buffering and allocation mechanisms found in advanced superscalar processors. However, limitations on instruction-level-parallelism [76] and the impact of a power limited era of VLSI design [60] have led to architects looking elsewhere for future performance improvements.

Designers are now moving to higher levels of parallelism to continue to deliver the processing performance demanded by consumers. Exploiting thread-level parallelism provides many exciting opportunities to accelerate single applications, provided that the algorithm can be decomposed into parallel threads. Such parallel programs have existed for many years. In the past, they have primarily been used for high performance computing applications, and were almost exclusively run on large multi-node systems. However with the advent of sub-micron process technologies, and the large transistor budgets they afford designers, it is now possible to integrate a multiprocessor system onto a single die.

2.1.1 Architectural Characteristics

Integrated multiprocessor systems are commonly referred to as chip-multiprocessors, or CMPs, and possess many interesting characteristics when compared to larger multi-node systems.

Communication Latency The close physically proximity of the computational units in CMPs allows data to be moved between them in a matter of cycles. Previously, signals between cores in multi-node systems would take hundreds of processor cycles; now even messages crossing the entire die can be expected to arrive within 2-3 cycles [28]. This dramatic shift in costs has become a defining feature of chip-multiprocessor architectures, and motivates designers to re-evaluate many of the established mechanisms used in multi-node systems.

Limited storage Although the integration of many cores on a single die has significantly reduced the latency of communication, it has also placed restrictions on the amount of storage immediately available to each processing core. In multi-node systems where each die might only contain a single processor, the entire on-chip storage budget could be allocated to provide large, dedicated caches. In CMPs, these same on-chip resources must be split between each of the processors on the die, reducing the effective cache size of each individual core. However this sharing of resources, while presenting many challenges to designers, also offers new opportunities for architectural innovation, particularly when cores are communicating via shared on-chip caches.

Fine-grain synchronisation The lower communication costs found in CMPs encourage software engineers to explore algorithms that may rely on finer-grained synchronisation mechanism than previously considered. Where large multi-node programs may rely on crude barrier synchronisation, CMPs are able to support fine-grained mechanisms that allow for faster synchronisation, and facilitate superior load-balancing.

Many groups throughout industry [5; 53] and academia [4; 73] have focused resources on developing new architectures, leading to a wide spectrum of designs. Presented below are a number of these designs, each showing how designers can use the huge number of transistors available to create efficient parallel compute platforms.

2.1.2 Research Architectures

2.1.2.1 MIT Raw

The Raw microprocessor was developed at MIT by Anant Agarwal et al. [73]. The goal was to design a general-purpose architecture that could exploit all levels of parallelism,

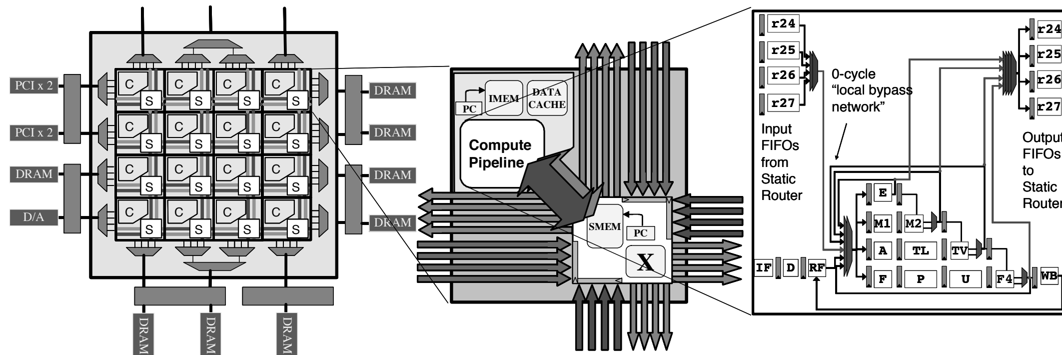


Figure 2.1: The Raw architecture, reproduced from original publication [73].

from instruction-level, to both data and thread-level parallelism. The architecture is a tiled multiprocessor, and features a very low latency network-on-chip that supports the distribution of operands across the chip. In the first work published, Argarwal et al. explored a version of the design with 16 tiles organised in a 4x4 mesh, although the system can scale to many more processors. The design exposes the communication fabric to the programmer through new ISA extensions, and each part of the architecture is designed with this goal in mind. Figure 2.1 shows the architecture at three levels of detail – chip level, tile level, and pipeline level.

Core architecture Each processing tile contains an 8-stage in-order single-issue processor, implementing a MIPS-style pipeline. Additionally, there is a single precision floating-point unit. Although each core is relatively small, the Raw architecture can scale to many hundreds of tiles, so the computation power quickly grows when sufficient parallelism is available.

Interconnect architecture Every tile contains routers to connect the processing elements to the four separate physical networks in the Raw architecture. The networks are each

32-bits wide and implement a 2D mesh topology, providing the high bandwidth needed to exploit the distributed compute resources available in such tiled architectures.

Interestingly, Raw employs both static and dynamically routed networks. The static networks are used when the source and destination of a message are known at compile time. Using the static network yields efficiency benefits, as there is no need to create or read packet headers at each router. Should the destination of a message be undeterminable at compile time, for example those relating to cache misses and interrupts, it is possible to use the two dynamic networks. These networks provide a deadlock free communication fabric for general packet switched traffic. Although the dynamic network features more complicated flow-control and routing mechanisms, the inter-tile latencies are still as low as 3 cycles.

Memory architecture The memory of the Raw processor is distributed throughout each of the tiles – there is no shared cache. Each tile contains a data cache (32KB) and two instruction caches. The first instruction cache (32KB) is used for processor instructions and the second cache (64KB) is used to store the instructions required by the static network routers. With each tile using only 128KB of cache, it is possible to create a system with many cores, without concern for the scalability of unwieldy shared caches architectures.

Prevalent features In particular, it is the novel interconnect architecture that makes Raw an intriguing design. The use of both static and dynamic networks, and the leveraging of the compiler in scheduling communications are all interesting approaches to optimising communication in a chip-multiprocessor system. The advanced communication fabric proposed, especially in comparison to simpler bus-based systems, creates

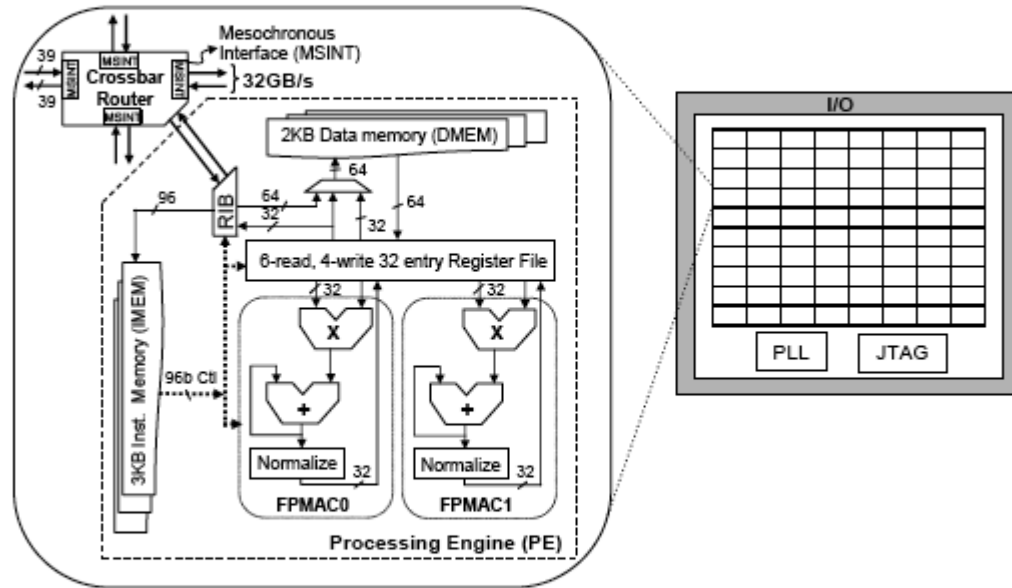


Figure 2.2: The Intel Polaris architecture, reproduced from Intel materials.

a scalable design that is well suited to the resource challenges hardware architects will soon be facing.

2.1.2.2 Intel Polaris

The Polaris testchip was developed by Intel Research [5] to demonstrate their ability to integrate a large number of compute cores, complete with network-on-chip fabric, into a single die. The chip delivers more than 1 teraflop of compute throughput – one trillion floating point computations per second – a remarkable level of performance for a single die. However the architecture is far from general-purpose and serves primarily as an engineering prototype to analyse the manufacturing, communication and power dissipation issues in a chip-multiprocessor system. Figure 2.2 shows the architecture at both the tile level – with core, memory and interconnect – and at the chip level – depicting a basic floorplan.

Core architecture Each tile contains two floating point units that implement a single precision floating point multiply accumulate operation. The cores do not support division, integer arithmetic or conditional jump statements, so cannot be used for true general-purpose computing applications.

Interconnect architecture Each tile in the system is connected via a high performance network-on-chip fabric to each of the adjacent tiles in the system, forming an 8 x 10 2D-mesh topology. The packet switched network operates at 4GHz [75] and features many advanced features to provide a total bisection bandwidth of 2 TB/s. This high performance interconnect fabric is used to provide the vast amount of data required to keep the 80 compute tiles busy.

Memory architecture Polaris uses a very small amount of local memory per tile - just a 3KB VLIW instruction memory, and 2KB data memory. However, this small allocation of memory allows the 80 cores of the system to fit onto a single 275mm^2 die. As there is a minimal amount of storage on-chip, it becomes vitally important for software to efficiently use the resources available.

Prevalent features The Polaris architecture is of interest largely due to its raw speed. The design shows that it is possible to get sufficient data onto a die to achieve extremely high floating point throughput. Although the design is limited by a lack of general purpose computation units, future designs at smaller technology nodes could use the extra transistors available to add such resources.

2.1.2.3 Intel Larrabee

The 32-core Intel Larrabee architecture was originally designed as an x86-based graphics accelerator, but development was halted in late 2009. The design has since been

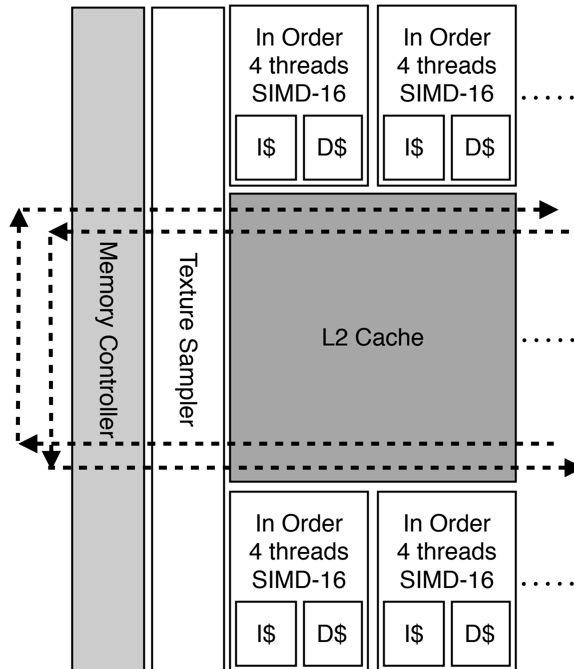


Figure 2.3: Intel Larrabee.

retargeted at high-performance computing applications, in an architecture code-named Knights Ferry. Although it is still under active development, the architecture provides an interesting datapoint in the chip-multiprocessor design space. Larrabee focused on supporting a new software-based graphics pipeline and was designed to ensure the architecture remained a powerful parallel processing platform for more general-purpose computation. Figure 2.3 shows four cores from the 32 core Larrabee system, the dotted lines representing the ring network used to connect all of the resources.

Core architecture Each core supports the Pentium x86 instruction set, and in addition, features extensions for managing caches and controlling a vector processing unit. The cores use an in-order pipeline, minimising the area required, and allowing many cores to be integrated on a single die. By supporting the entire x86 instruction set, Larrabee is

able to run existing code and operating system kernels, as well as making use of existing compiler technology.

Interconnect architecture The 32 cores in the Larrabee are connected to multiple bi-directional ring networks, providing scalable communication between all parts of the chip. The network is designed with minimal buffering, and instead employs basic static routing policies to avoid deadlock. As the cores contain reasonably sized caches, this simple network provides sufficient bandwidth, while keeping hardware overhead to a minimum.

Memory architecture The cores of the Larrabee employ a traditional two-level cache hierarchy; 32KB of data cache, 32KB of instruction cache, and 256KB of L2 cache. The L2 cache is shared between all cores in the chip, but cores have low-latency access to their local bank of this cache. This general-purpose memory hierarchy creates a flexible platform that can be used for many other applications beyond software programmable graphics pipelines.

Prevalent features Larrabee shows that it is possible to design an architecture which a high core count that still maintains legacy support for the important x86 code-base. It is also interesting that the proposed design uses a simple ring network, suggesting that if workloads remain sufficiently data-parallel, it may be plausible to avoid complex network-on-chip solutions.

2.1.3 Industry Architectures

The chip-multiprocessors presented thus far have all been the work of various research groups around the world. However, there are already a number of designs available

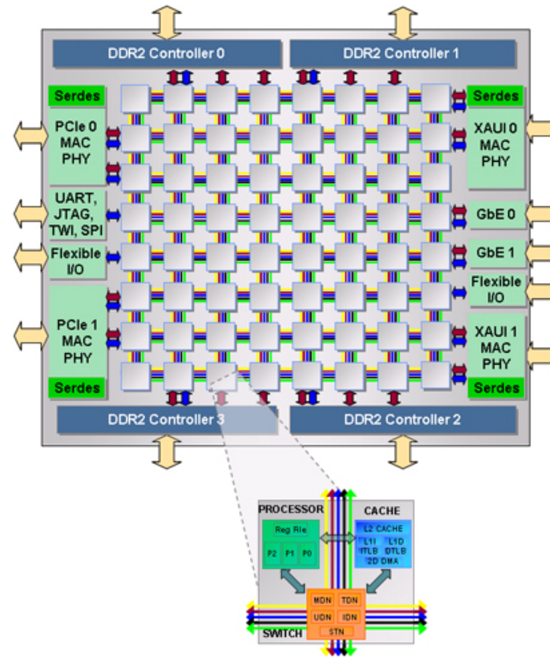


Figure 2.4: The Tilera Tile64 architecture, reproduced from original publication [11].

as finished products. Presented here are offerings from Intel, Sun and IBM, as well as Tilera, a productised version of the MIT Raw architecture described in Section 2.1.2.1.

2.1.3.1 Tilera Tile64

Following the success of the Raw project at MIT [73], Anant Agarwal founded a spin-off company to produce a product based on the Raw design. Tilera now offer a number of products based on the original Raw design, primarily focused at embedded processor markets such as networking and digital multimedia hardware. Figure 2.4 shows the architecture at both the chip level, and the tile level. The I/O interfaces are arranged around the perimeter of the chip, allowing easy access for packaging.

Core architecture The processors used in the Tile64 are more substantial than those found in Raw, employing a 3-way VLIW architecture with extensions for SIMD and

multimedia instructions. The platform support several programming languages, including full ANSI C, allowing legacy code to be easily ported to the system.

Interconnect architecture The cores are connected via five independent networks, each 32-bits wide. As in Raw, both static and dynamic networks are used, each serving a different class of traffic. The five networks are: the user dynamic network (UDN), I/O dynamic network (IDN), static network (STN), memory dynamic network (MDN), and tile dynamic network (TDN) [78]. The UDN and STN are used for user-level inter-tile communication. The MDN and TDN are used by tiles to communicate with memory controllers; the TDN carries requests, and the MDN responses. The IDN is used for I/O and OS-level communications. The interconnect is exposed to the programmer to provide extremely efficient operation when targeting streaming applications.

Memory architecture Each tile contains a traditional two-level cache hierarchy. The 8KB instruction and data caches are backed by a 64KB unified L2 cache, and TLB hardware for virtual-memory support. This memory architecture allows standard C programs and OS kernels to be compiled and run on a single tile, obtaining moderate performance.

Prevalent features The Tile64 is a productised version of the Raw architecture, so the most interesting part of the architecture is again the advanced interconnect system. However, it is also interesting that it has been possible to create a robust industry version of what at first was an academic research project. Tile64 proves there is a market for large scale chip-multiprocessor system-on-chip architectures.

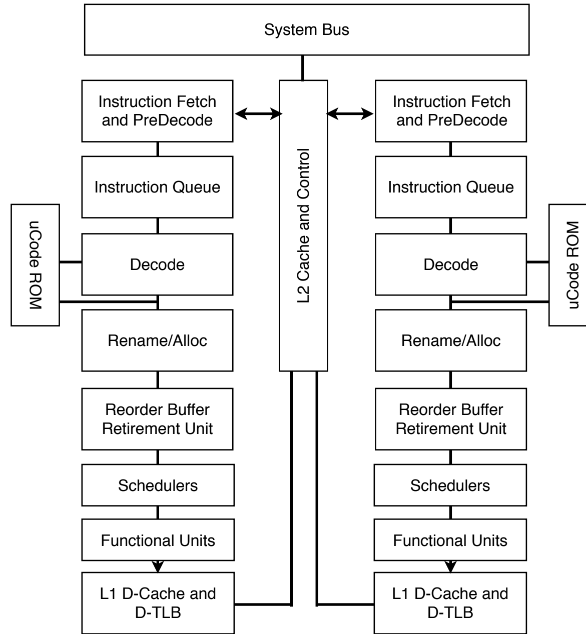


Figure 2.5: Intel Core 2 “Conroe” architecture.

2.1.3.2 Intel Core 2

Since its introduction in 2006, the Intel Core architecture has become the dominant commodity processing platform, found in the majority of desktop machines sold today. The design has gone through several iterations, but the basic concept remains – a small number of powerful, out-of-order x86 cores, attached to a centralised interconnection network. Figure 2.5 shows the dual-core Conroe system, one incarnation of the Core architecture.

Core architecture The latest Core architectures now contain up to eight processors, each supporting two simultaneous multithreading (SMT) threads. This technology allows two threads to share the resources of a single physical processor, increasing utilisation and helping to negate the fixed overhead of some of the complex buffering structures in advanced superscalar processors.

Interconnect architecture Early Core architecture products used the front side bus to communicate between cores – a simple but effective way of connecting a small number of cores, especially when running multiple single threaded applications. The newer products based on the Nehalem Core architecture now employ a proprietary point-to-point interconnect technology named QuickPath [81]. This is a more scalable solution that will allow core counts to continue to grow, even in the face of the increasing demands placed on the interconnect by truly parallel workloads.

Memory architecture To ensure that the powerful processors are kept busy, the architecture features very sizeable caches in each core. 32KB L1 instruction and data caches are backed by 256KB private L2 caches. Below the L2 there is a multi-megabyte L3 cache shared across all the cores in the system. This cache hierarchy is able to provide effective caching for both single and multi-threaded workloads, but requires a considerable portion of the transistors available on the die.

Prevalent features The Core 2 architecture is included here as it is the most popular desktop processor design in the market – it can be considered the incumbent chip-multiprocessor design. It represents a highly refined architecture, but it also carries the baggage of legacy support for the large x86 ISA.

2.1.3.3 Sun UltraSPARC T2

The UltraSPARC T2 multiprocessor, also known as the Niagara 2, was introduced in 2007, specifically targeting thread-parallel server workloads. The architecture is designed to deliver efficient, high throughput performance when exposed to large numbers of concurrent requests. By exploiting these high levels of parallelism, the cores can be kept busy during long latency memory accesses. Figure 2.6 shows the UltraSPARC T2

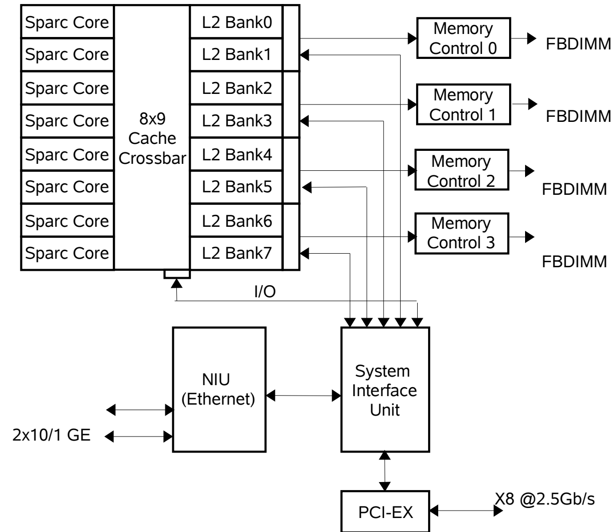


Figure 2.6: The Sun UltraSPARC T2 architecture, reproduced from original publication [55].

design, and illustrates the partitioning of resources in the system, in particular that the L2 cache sits the other side of a crossbar from all of the computation units.

Core architecture The UltraSPARC T2 contains eight SPARC ISA cores, each able to run eight threads concurrently using fine-grained multithreading technologies. Each core contains an 8-stage in-order pipeline. Little hardware is dedicated to complex branch prediction and prefetching; memory access delays are instead accommodated by switching in alternative threads. In line with the integer heavy nature of server workloads, the T2 contains two integer ALUs and only a single floating point unit per core.

These simple cores are efficient, but sacrifice single-threaded performance. Techniques such as hardware scouting [21] have been proposed to work around this limitation.

Interconnect architecture As there is little communication between threads in the server workloads that the T2 runs, the interconnect can be relatively basic. The eight cores are

connected to a pipelined crossbar switch that links the private L1 and shared L2 caches and although arbitration is required to access the interconnect, with the relatively small number of cores in the T2, the simple communication fabric is sufficient.

Memory architecture Each of the eight SPARC cores contain an 8KB L1 data cache and a 16KB instruction cache. Due to the 8-way multithreading used, the effective size of each thread's private cache is considerably reduced. This means that the operating system must employ techniques such as randomising the stack location in order to minimise conflicts between threads running on a single core. Below the modest private caches there is a similarly small shared L2 cache – 4MB, 16-way associative. Again, this cache must serve all 64 threads in the system, and hence can easily suffer from a high number of evictions due to address aliasing. The Solaris operating system minimises this effect by employing page colouring, and hashed cache indexing.

Prevalent features The UltraSPARC T2 is included here as it shows how fine-grained multithreading technologies can be used to improve efficiency of a parallel architecture designed for high throughput operation.

2.1.3.4 IBM Cell

The Cell Architecture was developed by IBM in collaboration with Toshiba and Sony [47] and is the only heterogeneous architecture presented here. The system is heterogeneous as it contains differently sized cores, one PowerPC ISA core and eight smaller co-processor units. This provides an interesting point in the CMP design space, and requires substantially different interconnect and memory architecture to those encountered so far. Figure 2.7 shows the hierarchy of resources in the Cell, and how the bus topology used to connect them .

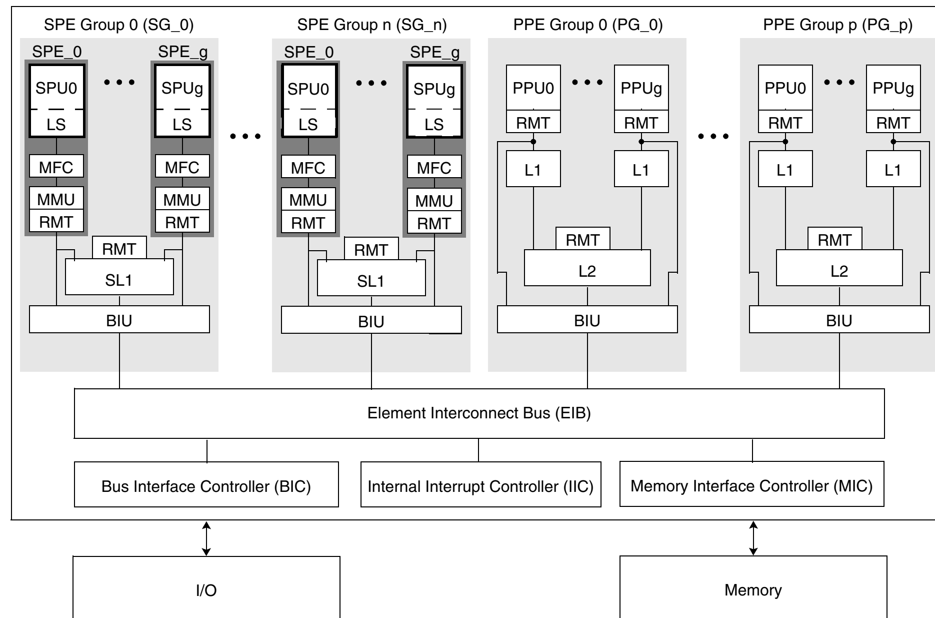


Figure 2.7: The IBM Cell BE architecture, reproduced from original publication [47].

Core architecture The single large core in the heterogeneous Cell architecture is known as the "Power Processor Element" or PPE. The PPE can run two threads using dual-issue, simultaneous multi-threading technologies, but is an in-order core, unlike the recent PowerPC architectures from IBM. This means that to achieve maximum performance, it is vital to effectively utilise the other compute resources available on the chip.

Located close to the main PPE there are eight small processors called Synergistic Processor Units, or SPEs. These processors use a different instruction set architecture and therefore require code to be compiled specifically to be run on SPEs rather than the PPE, presenting a major challenge to software engineers. The SPEs function as single-instruction multiple data (SIMD) engines, each issuing up to two in-order instructions per cycle. The pipeline is statically scheduled and due to the micro-architecture used, it is only possible to issue two instructions simultaneously if one is a compute operation and the other a memory operation. This static pipeline, combined with the lack of dynamic

branch prediction, means that the compiler must provide substantial assistance to make efficient use of the SPEs.

Interconnect architecture The eight SPEs and single PPE are connected via the proprietary bus known as the Element Interconnect Bus (EIB). The I/O interfaces and the memory interface controller are also attached to this bus.

The EIB is in some ways similar to the ring network used in the Intel Larrabee architecture presented in Section 2.1.2.3. The links are bi-directional so the longest path between two units is half the size of the ring, keeping communication latencies to a modest number of cycles. When facing high contention, providing that the sharing patterns are suitable, the EIB can support several bus transactions each cycle, helping to keep all of the compute units in the Cell busy.

Memory architecture Reflecting the asymmetrical partitioning of compute resources in the Cell architecture, the memory hierarchies used for the PPE and SPEs are markedly different.

The PPE is backed by two levels of private cache – a split L1, 32KB for data and 32KB for instructions, and a combined 512KB L2 cache. This traditional cache architecture is used to support the conventional sequential portions of programs running on the PPE.

The SPEs do not use caches and instead each use 256KB of local scratch pad SRAM. This memory requires direct management from the software, placing burden on the programmer to ensure that SPEs are working primarily on data already present in the scratch pad.

Prevalent features The Cell architecture is of interest as it uses heterogeneous to improve efficiency of workloads with asymmetrical parallelism. It shows that through the

use of a novel memory architecture, it is possible to augment a traditional processor architecture (PowerPC) with coprocessors specialised for SIMD operations. The combination of these two elements produces a powerful parallel processing architecture, albeit at the expense of a simple programming model.

2.2 Interconnection Networks

The communication performance between compute nodes in a parallel system has a large impact on system performance and this is increasingly true as more applications rely on fine-grained communication to parallelise previously sequential tasks. Combined with the growing number of processing elements, this places great demands on the communication fabric in modern chip-multiprocessors. Existing interconnection systems can struggle to scale and meet these demands, which has lead to an increasing amount of research and design effort being spent investigating these issues.

2.2.1 Bus-based Interconnects

In recent times, the most prevalent forms of interconnect network have been shared bus fabrics. This simple technology is most frequently used to connect the many parts of a system-on-chip system. Bus systems such as AMBA [34] and CoreConnect [13] have been accepted as industry standards, allowing for a variety of components to be easily connected to form powerful, customised systems. Beyond these bus standards, low core-count parallel architectures often employ proprietary forms of bus-based interconnect

Architecturally, a bus behaves as a set of wires shared between all client nodes. This is illustrated in Figure 2.8. Centralised arbitration is used to ensure that the shared interconnect is written to by only a single node at one time.

This scheme has both advantages and disadvantages. First, the central arbitration

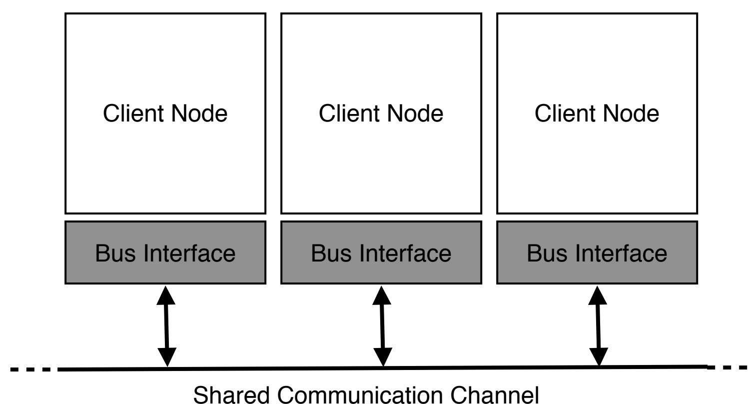


Figure 2.8: A generic bus architecture.

provides strong ordering between requests that is a natural fit for the requirements of cache coherence described in Section 2.3. Additionally, all nodes can listen to the transactions on the bus, allowing global system knowledge to be maintained at each private client node. In situations with little contention, bus systems can provide very high performance at a minimal hardware cost.

However bus communication fabrics have several drawbacks. Under high contention – caused by heavy traffic patterns or a high number of clients – buses will soon become a bottleneck in a parallel system. Furthermore, the physical distance between communicating nodes has grown considerably in line with the large die sizes now fabricated. This means that the shared wires have to stretch across the entire chip, creating a large capacitive load and pushing up power consumption. These challenges have forced designers to look at new ways to support communication in chip-multiprocessors.

2.2.2 Network-on-Chip

To combat the performance and power issues of shared bus interconnects, designers now implement embedded network-on-chip systems to provide scalable, efficient com-

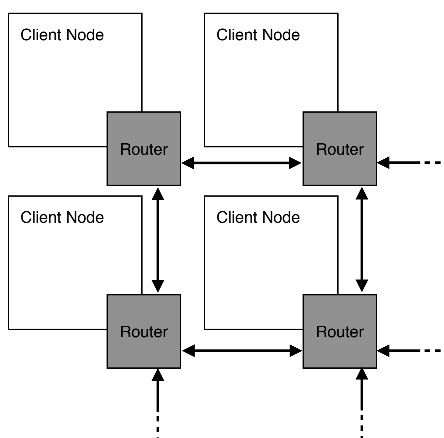


Figure 2.9: A generic network-on-chip architecture.

munication [28; 61; 75]. These systems employ distributed routers connected by a large number of individual wiring channels, and avoid many of the scalability issues of buses. Figure 2.9 depicts a generic example of one such architecture. Distributed arbitration allows for simultaneous communication events in the interconnect, supporting greater concurrency and alleviating contention. Similarly, the greater number of wiring channels improves the cross-sectional bandwidth by allowing messages to be transmitted over a number of shorter links.

2.2.3 Flow-control Mechanisms

With the large number of resources required for a network-on-chip including buffering, crossbars and wiring channels, it is important to implement schemes to fairly and efficiently allocate these resources to communication requests. The goal of this allocation is to manage the rate at which data is transmitted from sender to receiver. This task of controlling data rate is referred to as flow-control.

The challenges of flow-control have been addressed previously in the field of full scale networks. Systems such as TCP/IP [6] and UDP [68] are good examples of network

specifications that incorporate complex flow-control schemes. On-chip designs are more constrained and must use pared down versions of the concepts to meet power and area budgets [28; 61].

On-chip flow control designs can be split into two broad categories: circuit-switched and packet-switched. Both schemes are well studied and offer many advantages. However traffic profiles and performance expectations will usually dictate the technique utilised for a design.

In a circuit switched network [44], a routing path is established between source and destination nodes before any data is sent. Any resources required along the path are reserved and then deallocated once the transmission is complete. To improve the total number of concurrent circuits supported by a system, the resource allocation is often time-multiplexed to allow the sharing of a single physical resource.

Thanks to the lack of per-packet arbitration, circuit-switched networks provide a very high bandwidth connection once the initial allocation has been performed, making them ideally suited to applications with very high throughput communication between nodes.

Unfortunately, during the time taken for the header packet to traverse the network and allocate resources, no data can flow between source and destination nodes. This can have an adverse affect on many latency sensitive applications and can prohibit the use of circuit switched flow control if the traffic consists of short, unpredictable transactions.

However, it is not necessary for circuits to be established dynamically at run-time. The communication paths can be statically scheduled at compile-time, removing the requirement for the long latency resource allocation stage. Data is allowed to flow freely between nodes at statically defined times. In many embedded applications the traffic patterns are sufficiently defined to be amenable to this static scheduling technique — a property exploited by architectures like the Tilera CMP [11].

In contrast, packet-switched flow control systems [28] allocate resources dynamically on a per-packet basis, greatly increasing resource utilisation for unpredictable traffic patterns. Messages are split into several packets, and buffering is added to routers to allow for the temporary storage of delayed packets. The use of a dynamic flow control technique creates many opportunities to interleave packets of different messages to improve utilisations of the underlying physical resources.

By allocating resources dynamically at each node, packet-switched networks can provide higher performance for a variety of traffic patterns. However, adding buffers and increasing the complexity of resource arbitration introduces additional design challenges in maintaining fairness and forward progress. More complex solutions such as virtual-channel flow control [61] and back-pressure buffer control mechanisms have been used to combat these issues.

For general-purpose computing applications it is widely acknowledged that it is necessary to have at least some form of dynamically allocated network resource. This allows a single network design to support a wide variety of run-time applications – an essential characteristic of any general-purpose architecture.

2.2.4 Low-Latency Router Design

The most advanced network-on-chip designs built to date are complex packet-switched systems that deliver extremely low-latency arbitration and routing [75]. Intel developed the Polaris test-chip (Section 2.1.2.2), in part to explore the challenges surrounding the manufacturing of such interconnection systems.

As previously discussed, packet switched networks are ideal for general-purpose parallel platforms with large numbers of compute nodes, but due to the complex arbitration and allocation phases, many early on-chip networks would employ deeply pipelined

routers [66]. For latency sensitive applications this is far from desirable, and designers began to focus on developing routers that could support higher clock frequencies, while employing the minimum number of pipeline stages.

New designs emerged to achieve these goals. Of particular note, work by Mullins et al. [61] exploited speculative techniques in resource allocation that drive down router delays to provide a single-cycle common case latency. Such designs can consume a considerable amount of power, but this is justified by the importance of low latency inter-node communication, especially in cache coherent systems.

2.3 Cache Coherence

A key challenge to creating scalable parallel computing platforms, while not sacrificing programmability, is to maintain suitable models of coherence and consistency. Both of these properties are related to the ordering of memory operations, and subscribing to the varying levels of coherence and consistency can have a dramatic effect on the performance of a system.

To trade-off of between complexity and performance, designers have proposed a great number of coherence protocols and to establish the spectrum of designs available, this section presents examples of centralised and distributed cache coherence protocols.

2.3.1 Consistency

The first consideration is that of memory consistency. The introduction of out-of-order execution and advanced load-store queues has had an important impact on the behaviour of memory operations in the system. In older systems, memory operations would commit in program order, providing predictable behaviour. However, it is now rare to

find such consistency in new architectures. This section outlines the three most common models encountered – sequential, weak, and release consistency.

Sequential consistency [52] is achieved when"the result of any execution is the same as if the operations of all the processors were executed in some sequential order, and the operations of each individual processor occur in this sequence in the order specified by its program". This effectively places two bottlenecks in the system — the first at the level of each individual processor, where loads and stores cannot be re-ordered, and second, at the level of the shared global memory, where all memory accesses must pass through a single ordering point. Regrettably, although sequential consistency is attractive from the perspective of software engineers, the restrictions placed on the hardware design mean that designers often look to less strict models of consistency. However, as put forward in work by Hill [39], there is still a strong case for keeping the simplest possible consistency model.

Designers soon realised that it is unnecessary to place strict ordering constraints on all memory accesses in a system. Work by Dubois et al. [30] described the model of Weak Consistency, in which sequential consistency is only applied to synchronisation variables. Weak Consistency demands that any previous memory accesses are satisfied before a synchronisation variable can be accessed, and in this way provides a useful model to programmers, while still allowing hardware optimisations to re-order memory operations on regions of shared data.

In some cases, even Weak Consistency can be too restrictive, and designers have found that by splitting synchronisation operations into two phases, it is possible to further increase the permissible memory operation re-orderings. The first phase is importing information, such as acquiring a lock, and the second phase is exporting information, such as releasing a lock. Gharachorloo et al. [35] developed Release Consistency based on these two primitive operations. The restrictions placed on memory operation

orderings are somewhat similar to those found in Weak Consistency. First, all normal load and store operations must be completed before an acquire access can be performed. Second, all normal load and store operations must be completed before a release operation can be performed. Finally, both acquire and release operations must be sequentially consistent with respect to each other. Release Consistency has been widely adopted as a powerful but programmable model. It is found in both Java [36] and OpenMP [20].

Consistency models are still a very active area of research, with designers looking for new ways to relax memory ordering requirements, while maintaining a simple and intuitive interface for software engineers. The examples presented here merely represent the most commonly encountered designs that are relevant to the protocol work in Chapters 5 and 6.

2.3.2 Coherence

Achieving coherence in a system can be formally defined as“the results of any execution of a program are such that, for each location, it is possible to construct a hypothetical serial order of all operations to the location”. As described by Culler and Singh [27], this implies two properties. The first is write propagation – all writes become visible to other processes. The second is write serialisation – all writes to a location are seen in the same order by all processes. These properties are of vital importance if the system is to deliver predictable performance.

Although this problem is unique to parallel systems, it is not a new one. There is a long history of designs that address this challenge, but not all are ideally suited to new chip-multiprocessor architectures. In particular, the interconnect used has a strong influence on the coherence mechanisms that can be used, and with newer network-on-chip based designs emerging, now is an ideal time to re-evaluate this field.

2.3.3 Memory Models

A fundamental decision in the design of any chip-multiprocessor is the memory model to use. At the highest level the choice is between distributed memory message-passing and shared memory systems.

Distributed memory systems use explicit methods of communication between nodes, such as message passing. In shared memory systems, a communication event is not explicit, it is instead described by a particular pattern of accesses to a shared global address space. Both architectures have their advantages.

Distributed architectures provide clear partitions in an algorithm and require the programmer to fully consider the implication of any communication between nodes. This is particularly desirable with the rising cost of communication seen in modern process technologies, and encourages programmers to find ways to minimise the data transferred between discrete processors. Additionally, as there is a clear divide between local and remote memory accesses, there is zero additional overhead for accesses to private regions of memory. Unfortunately there are drawbacks that have led to message passing systems being used almost exclusively for supercomputing applications. The greatest challenge is the transition from uniprocessor programming to a fundamentally different way of thinking about computing. Using distributed memory requires a new approach to algorithm design that has led to message passing systems being used only by a small number of highly skilled experts. Furthermore there are technical challenges, such as the migration of processes between nodes. If it becomes necessary to move computation to a physically remote node, it is required not only to move the instruction memory, but also all private data needed to continue the process at the new location. This additional level of complexity makes message passing less desirable in situations where dynamic load balancing is essential.

Shared memory systems provide a single global address space, often with no concept of explicitly private areas of memory for each node – any address can be accessed from any node in the system. The advantage of such an organisation is that, from the programmers' perspective, the system behaves like many cores attached to a uniprocessor memory subsystem. This helps keep the programming model close to the uniprocessor systems that many programmers are familiar with. Furthermore there are large benefits when considering the amount of legacy single threaded code that can be run on a shared memory system without the need for major refactoring. While shared memory systems are simple from the programmers' perspective, the overheads are found in the complex hardware mechanisms needed to maintain this abstraction. When hiding the cost of communication from the programmer, hardware designers must intelligently design protocols to provide efficient chip-multiprocessor memory architectures.

Such systems have been in development for many years. Work by Kai Li on the IVY shared memory system [56] put forward the case for shared virtual memory systems over message passing protocols. However the mechanisms by which this model is maintained is still an active area of research.

2.3.4 Snooping and Directory Systems

When using a communication architecture that allows all participants to observe all messages transmitted, it is possible to implement coherence with a simple snooping protocol. Such protocols were used in early multi-processor systems that employed shared-bus interconnects [65]. It is also possible to design network-on-chip architectures that support broadcast [46], and therefore also support snooping, but these are less common.

Snooping refers to the mechanism by which each processor observes the reads, and most importantly, the writes issued by all other processors in the system. This allows

each processor to check the address of a snooped memory access against its local cache contents, and update or invalidate the local data as dictated by the protocol. This mechanism normally leads to coherence being maintained at a cache line granularity, although some researchers have proposed tracking accesses at a coarser granularity [18; 33].

This whole process is facilitated by the expansion of each cache line to hold a coherence state, in addition to the data. The number of different states used by the coherence protocol differs according to the cache architecture, and the extent to which the designer wishes to reduce traffic across the shared interconnect.

A defining feature of a snooping system is that each node holds the state necessary for coherent operation, and maintains the state by watching memory traffic on the interconnect. A different solution is to maintain sharing state in a separate entity called the directory. The directory contains the sharing information of each cache-line held privately anywhere in the system.

The most intuitive solution is to use a single physical resource through which all memory requests must pass; if adequate state is maintained at this location, it is possible to meet the necessary ordering requirements. This directory can reside in a number of locations in the memory hierarchy, but the function remains the same — any memory request must first be delivered using the point-to-point interconnect to the directory where the appropriate coherence messages are generated.

Directory protocols have appealing characteristics: low bandwidth requirements, and a good match to the architecture of lower levels of shared cache. However keeping a single directory structure means that, as with the crossbar, contention encountered under high loads will quickly degrade performance.

2.3.5 Distributed Systems

While shared communication fabrics such as crossbars are adequate for small numbers of nodes, they soon become a bottleneck when scaling beyond eight cores. For example, the Sun Niagara processor described in Section 2.1.3.3 uses eight cores with an advanced crossbar interconnect. However, new Intel designs using upwards of eight cores will now use point-to-point communications – the QuickPath protocol [81].

This shift to point-to-point communication layers requires that cache protocols no longer rely on broadcast and simple snoop or directory mechanisms to maintain coherence. Instead, the protocols must use some form of distributed ordering point, be it physical or virtual, to ensure that the coherence and consistency properties are fulfilled. In particular, distributing a directory around the system solves this while providing additional benefits.

A common method for partitioning the monolithic directory structure is to create many smaller directories, each responsible for a portion of the address space. Interleaving the address lines across the directories provides load balancing, helping to avoid hotspots and contention in the communication layer. These concepts were first proposed by Lenoski et al. for use in the Stanford Dash Multiprocessor [54], and have been widely accepted as a promising technique to maintain coherence across a distributed interconnection network.

Distributed systems do, however, sacrifice any simple way of supporting write-update protocols. This means that shared data is invalidated across the chip when updated by a single processor, and hence leads to longer latency core-to-core communication should the value need to be read remotely. Cheng et al. published work detailing mechanisms by which write-updates can be supported [22] but almost all systems using distributed directories use a simpler write-invalidate scheme. Further complications are

encountered due to the storage overhead of directory structures in large systems. Researchers have proposed methods to reduce this burden by rearranging the cache tag architecture, providing scalability up to 1024 nodes [80].

2.3.6 Protocols

Examples of the trade-offs available to designers can be drawn from the evolution of the *VI*, *MSI* and *MESI* protocols.

The *VI* protocol is the most basic solution, in which there are two states — valid (*V*) and invalid (*I*). The behaviour of the protocol is straightforward — when a processor reads or writes data, the state is set to valid. The data then remains valid until a write from another processor occurs, at which point one of two things can happen. In a write-update protocol, the resident local data will be updated with the new value from the write, or in a write-invalidate protocol, the data will be moved to the invalid state and must be loaded from memory if it is needed again. In this simple protocol, there is no advantage to using a write-invalidate protocol as all writes must be propagated across the interconnect to maintain coherence.

Although the *VI* protocol provides coherent operation, it does nothing to reduce the amount of traffic on the interconnect, so scalability becomes a major problem. Consider the large number of reads and writes that are operating on private data. In these situations it is not strictly necessary to update other nodes after all accesses, providing that sufficient state is maintained to ensure coherent operation when memory operations do communicate between processors. A simple extension can ensure that in most cases, write accesses are only propagated when the data is shared elsewhere in the system.

The *MSI* protocol uses three states – modified (*M*), shared (*S*) and invalid (*I*). Invalid represents the same state as in the *VI* protocol, but there are now two valid states. Data

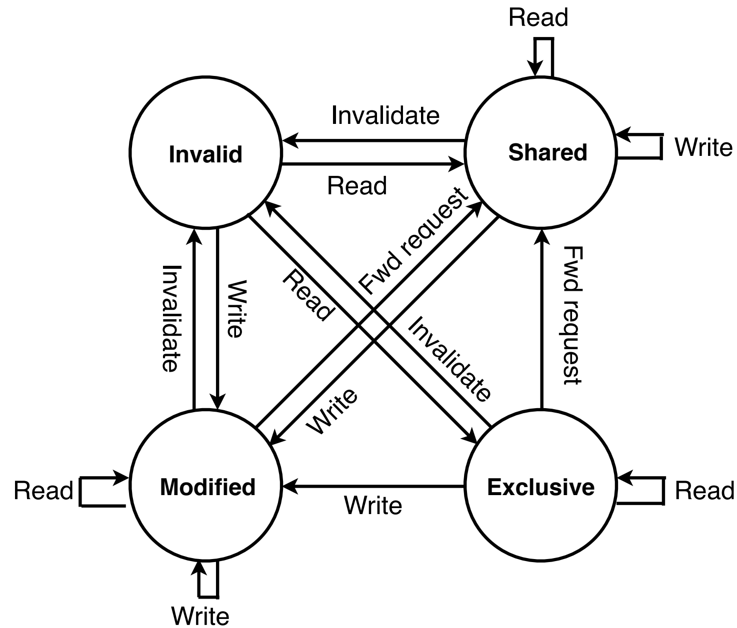


Figure 2.10: A basic MESI protocol state diagram.

is moved to the modified state if it has been written to by the processor. In this state, the local node can read and write to the location without initiating further transactions to remote nodes in the system. This is possible as the protocol guarantees that when a node holds a line in modified state, it does not exist elsewhere in the private caches of remote nodes. Should another node wish to read the line, the cache controller will downgrade the state of the modified data to shared. This simple modification greatly improves the bandwidth utilisation of the protocol.

A further optimisation is possible when considering that fresh data that is read and brought into the system by a particular node will often be written to in subsequent instructions by the very same node. Under the *MSI* protocol this would result in two transactions – the first, loading the data and placing it into the *S* state, and the second, a request to upgrade of permissions to the *M* state. By placing the data into a new exclusive state (*E*) when it is first loaded, these extra protocol transitions are avoided.

Doing so reduces both the latency of write operations, and the interconnect bandwidth required. This is the *MESI* protocol depicted in Figure 2.10.

Each of these protocols provides coherent operation between nodes sharing an interconnect fabric. There is a clear trade off between protocol complexity, and latency or bandwidth requirements. Importantly, each protocol advance is driven by easily measurable program behaviours.

Real implementations of these protocols use split-transactions to provide sufficient concurrency in the protocol. When using a shared communication channel, additional complexity must be added to support several outstanding requests from many client nodes; this is achieved by breaking down each request into several decoupled transactions (split-transactions). In most cases this will be a request and response message. Doing so adds additional states to the simple finite state machines covered thus far as the system must now describe behaviour for each node if it receives an unrelated request while an existing transaction is pending. Although this more than doubles the number of states required to maintain coherence, it is almost essential. Without support for split-transactions, a system can only support one active request, no matter the number of nodes, severely limiting scalability.

2.3.7 Summary

Cache coherence has long been a challenge to designers of parallel systems. Furthermore, the additional design constraints around interconnect and synchronisation performance complicate the task.

There is an extremely large body of research exploring ideas well beyond the basic protocols presented here, but the work in this thesis is based on a distributed version of the MESI protocol. More advanced protocols of relevance are discussed in Chapter 7.

2.4 Parallel Benchmarks

With both industry and academic institutions now increasingly interested in parallel computing, a substantial amount of effort is spent developing suitable benchmarks suites with which to evaluate future architectures. This task is non-trivial, first and foremost due to the challenge of predicting the ways in which future computing platforms will be used.

The motivation is clear – hardware designers must, especially in an era of power constrained design, develop systems specifically to cope with the changing demands of future applications. By evaluating the performance of systems running realistic parallel programs, hardware designers can make informed decisions to improve performance while remaining within increasingly challenging power envelopes.

Parallel benchmark suites have existed for some time [71], but have often been restricted to the domain of scientific high performance computing. These applications can typically be partitioned with ease, allowing them to exploit high levels of data parallelism. However, parallel architectures are now expected to run increasingly complex programs with large amounts of synchronisation between threads. Scientific compute performance is still an important consideration but old benchmark suites no longer represent the common usage of parallel processors [15].

In contrast, many new parallel benchmark suites include a variety of application domains, from multimedia processing to data mining [15]. The individual domains of interest have begun to mature and stabilise, but the algorithms to be employed are still under constant development [16]. For this reason, it is important to avoid tuning architectures to benchmarks employing soon to be obsolete software techniques. Research in this area has focused on characterising workloads at a more fundamental level, that of individual compute kernels. The most recognised work in this field has come from

Berkeley [4], where researchers identified several compute "motifs", each capturing a pattern of computation and communication. Initially there were seven basic patterns: dense linear algebra, sparse linear algebra, spectral methods, n-body methods, structured grids, unstructured grids and map-reduce. The list has since been expanded to thirteen patterns. It is hoped that by abstracting away some of the levels of complexity and considering more ubiquitous compute kernels, designers can create architectures that are less sensitive to the exact details of the algorithms employed.

2.4.1 Shared Memory Synchronisation Methods

All parallel benchmarks require a mechanism through which the multiple threads of each program can communicate. The exact method chosen is influenced by the underlying memory architecture as well as the algorithms used by the program. When considering shared memory architectures, the choice is usually between the use of mutex locks and barriers.

Mutexes (mutual exclusion) are designated variables for controlling access to regions of shared memory. They can be implemented in a variety of ways, often using underlying locking primitives, but from a software perspective provide two operations: acquire and release. When a thread acquires a mutex it is guaranteed exclusive access to the associated region of shared data. Once inside a region of code guarded by a mutex, execution can continue with no risk of data races between other threads. When the guarded instructions have finished executing the thread then releases the mutex, allowing competing threads access to the shared region of memory.

Mutex mechanisms allow threads to co-ordinate computation and communication. However, there are many intricacies relating to forward progress and fairness. Although these concerns are of great importance when writing parallel algorithms, my research

uses existing software benchmarks, hence the specifics of these matters are not discussed further.

Barriers are another common synchronisation method used in a variety of parallel programming applications. They are used when it is necessary for a certain set of threads to reach a particular point in their control flow before proceeding. A typical use is in scientific simulations with discrete time steps; all threads must finish local computation on the current time step before exchanging data and moving to the next iteration. Barriers are usually implemented in software using a combination of memory locks to provide the desired behaviour, although some new architectures propose hardware support for this operation to improve efficiency [19].

In an effort to improve programmability, researchers have developed more elaborate synchronisation mechanisms such as lock free data structures [8] and transactional memory systems [38]. Although these designs offer many benefits, their specifics are beyond the scope of the work in this thesis.

2.4.2 SPLASH-2

The SPLASH-2 suite of parallel benchmark applications was released in 1995 by researchers from Stanford and Princeton universities. The suite contains a variety of high performance computing and graphics applications, representative of the dominant parallel workloads of the time, and since its release has allowed designers to use representative programs for system analysis.

The suite contains twelve applications, briefly described below. Further details can be found in the original publication by Woo et al. [79]. For ease of reference, Table 2.1 is reproduced below, giving a overview of the communication behaviour of each program.

Barnes simulates the use of the Barnes-Hut N-body method to solve 3D particle inter-

Code	Problem Size	Total Instr (M)	Total FLOPS (M)	Total Reads (M)	Total Writes (M)	Shared Reads (M)	Shared Writes (M)	Barriers	Locks	Pauses
Barnes	16K particles	2002.79	239.24	406.85	313.29	225.05	93.23	8	34648	0
Cholesky	tk15.0	539.17	172.00	111.86	28.03	75.87	23.31	3	54054	4203
FFT	64K points	34.79	6.36	4.07	2.88	4.05	2.87	6	0	0
FMM	16K particles	1250.02	423.88	226.23	38.58	217.84	30.10	20	28088	0
LU	512 x 512 matrix, 16 x 16 blocks	494.05	92.20	104.00	48.00	93.20	44.74	66	0	0
Ocean	258 x 258 ocean	379.93	101.54	81.89	18.93	80.26	17.27	364	2592	0
Radiosity	room, -ae 5000.0 -en 0.050 -bf 0.10	2832.47	—	499.72	284.61	261.08	21.99	10	231190	0
Radix	1M integers, radix 1024	50.99	—	12.06	7.03	12.06	7.03	10	0	124
Raytrace	car	829.32	—	208.90	79.95	159.97	22.22	0	94471	0
Volrend	head	754.77	—	152.19	59.57	81.93	3.07	15	28934	0
Water-Nsq	512 molecules	460.52	98.15	81.27	35.25	69.07	26.60	10	17728	0
Water-Sp	512 molecules	435.42	91.50	72.31	32.73	60.54	22.64	10	353	0

Table 2.1: A breakdown of instructions executed in a 32 processor machine running SPLASH-2, reproduced from original publication [79].

action problems. *Cholesky* factors a sparse matrix into the product of a lower triangular matrix and its transpose. *FFT* computes the 1D FFT of a set of complex points using an optimised algorithm. *FMM* simulates a 2D N-body problem using the Fast Multipole Method. *LU* factors a dense matrix into the product of lower and upper triangular matrices. *Ocean* simulates ocean movements using a sub-grid decomposition. *Radiosity* lights a 3D scene using an iterative hierarchical diffuse radiosity method. *Radix* performs an integer radix sort on a set of keys. *Raytrace* renders a 3D scene using a hierarchical grid raytracing algorithm. *Volrend* renders a 3D scene of voxels using a ray casting technique. *WaterNsquared* evaluates interactions between water molecules using an $O(n^2)$ algorithm. *WaterSpatial* evaluates interactions between water molecules using a spatial subdivision technique.

Despite its age, the SPLASH-2 suite has remained extremely popular as a way to compare new architectures to previous designs. However, for evaluating newer designs, recent publications [15] have suggested that many of the algorithms are now out-dated, largely due to the increasing dominance of the CMP as a parallel computing platform and the new communication opportunities present in such systems.

2.4.3 Parsec

The Parsec benchmark suite was released by Bienia et al. in 2008 [15] and provides a selection of modern applications for use in architectural design and analysis. The drastic reduction in the latency cost of inter-core communication has been taken into account during the design of the algorithms used in Parsec. The benchmarks target a variety of application domains and, as with SPLASH-2, the programs are briefly described here for reference. As a further reference, a qualitative overview of the communication characteristics has been reproduced in Table 2.2.

Blackscholes is a financial simulation evaluating the Black-Scholes partial differential equation for the calculation of stock prices. *Bodytrack* is a computer vision application that tracks a human body in 3D space from multiple 2D images. *Canneal* is a cache-aware simulated annealing kernel used to minimise routing distances in an ASIC place-and-route operation. *Dedup* is a data compression algorithm used in enterprise storage systems. *Facesim* animates a human face using detailed physical simulations of underlying muscles. *Ferret* is a content-based image similarity search. *Fluidanimate* animates a fluid using smoothed particle hydrodynamics. *Freqmine* is a data mining application used for frequent itemset mining. *Raytrace* is a rendering application used to produce high fidelity images of a 3D scene. *Streamcluster* is a kernel to solve the online clustering problem in a stream of data. *Swaptions* is a simulation employing the Heath-Jarrow-Morton method to calculate the value of swaptions. *Vips* is an image processing system featuring a variety of parallelised transformations. *X264* is a video encoder.

Importantly, the suite includes a number of benchmarks that spawn more threads than the number of cores available, leaving the operating system to schedule work in an effective manner. The characteristics of each of the programs are presented in the original publication [15] from Bienia et al.

Program	Application Domain	Parallelisation Model	Parallelisation Granularity	Working Set	Data Usage - Sharing	Data Usage - Exchange
blackscholes	Financial Analysis	data-parallel	coarse	small	low	low
bodytrack	Computer Vision	data-parallel	medium	medium	high	medium
canneal	Engineering	unstructured	fine	unbounded	high	high
dedup	Enterprise Storage	pipeline	medium	unbounded	high	high
facesim	Animation	data-parallel	coarse	large	low	medium
ferret	Similarity Search	pipeline	medium	unbounded	high	high
fluidanimate	Animation	data-parallel	fine	large	low	medium
freqmine	Data Mining	data-parallel	medium	unbounded	high	medium
raytrace	Rendering	data-parallel	medium	unbounded	high	low
streamcluster	Data Mining	data-parallel	medium	medium	low	medium
swaptions	Financial Analysis	data-parallel	coarse	medium	low	low
vips	Media Processing	data-parallel	coarse	medium	low	medium
x264	Media Processing	pipeline	coarse	medium	high	high

Table 2.2: A qualitative description of the benchmarks in the Parsec suite, reproduced from original publication [15].

2.5 Summary

Chip-multiprocessor designs are rapidly evolving to meet the demands of modern application domains. This background chapter describes the state-of-the-art in a variety of aspects of parallel system design. For the work of this thesis, it is important to establish a suitable baseline system for experimentation, so for each of the many concepts presented here, it is necessary to select the most appropriate design for a modern, general-purpose chip-multiprocessor.

2.5.1 Processor Architecture

Cores of all sizes have been used in existing parallel architectures, ranging from the large cores of a consumer Intel product, to the small cores of a Tilera system. Future systems are likely to cover an even broader spectrum, with proposals for chip-multiprocessors using extremely simple cores [49] becoming increasingly prevalent. For a general purpose system, much of the processor architecture is dictated by the necessity for legacy ISA support. In most situations this means the use of x86 cores, although stringent

power budgets are now moving designers towards more efficient in-order architectures over power hungry superscalar implementations.

Longer term, designers face an interesting trade-off between parallel throughput and single threaded performance. This debate is far from settled but already many researchers believe that there will be a continued need for both aspects of parallel performance [32; 40].

2.5.2 Cache Coherence

Considering the great majority of programmers have worked exclusively on single-processor systems, to ease the necessary transition to parallel programming it is advantageous to use a memory model familiar to these programmers — the most logical choice being a shared-memory abstraction. In particular, many programmers are most comfortable with the concept of a cache-coherent shared memory.

Maintaining coherence at high core counts presents a number of challenges to designers, ranging from communication latency to power overhead. This thesis investigates how to maintain the shared-memory abstraction while minimising the overheads.

2.5.3 Parallel Benchmarks

In recent years a variety of new application domains have emerged. The vast amount of data generated by modern computer networks has led to the rise of data mining and search applications; a growing interest in advanced human computer interaction has lead to a boom in computer vision applications, and modern scientific research often relies heavily on simulations of increasingly complex systems using newly developed algorithms.

While these new applications are essential benchmarks to consider when designing

future parallel systems, there is still a very large library of existing applications that require continuing support. For this reason, this work analyses the performance of two benchmark suites: SPLASH-2, representing the large number of existing parallel applications, and Parsec, containing emerging workload domains.

2.5.4 Interconnection Networks

With the increased compute performance afforded by growing numbers of cores in a single chip, the bottleneck quickly becomes the communication fabric connecting the nodes. Existing bus-based technologies are suitable for designs with only a few integrated cores, and already systems with more than 16 cores have moved to more sophisticated interconnect solutions. This work assumes that this trend will continue, and that network-on-chip architectures will soon be common place.

Low-latency packet-switched routers offer enough performance to provide sufficient work to arrays of even the powerful x86 cores proposed for this work. Although it is common to pick a more conservative network, this work assumes that future architectures will require low latency interconnect systems to support fine-grained communication, and that such networks represent the only realistic baseline.

CHAPTER 3

Communication Characterisation

This chapter thoroughly examines the crucial communication and sharing behaviour of the programs that future processor architectures will be expected to run, encompassing both legacy and emerging application domains. The infrastructure used allows both accurate and comprehensive program analysis, employing a full Linux OS running on a simulated 32-core x86 machine. Experiments use full program runs, with communication classified at both core and thread granularities. Migratory, read-only and producer-consumer sharing patterns are observed and their behaviour characterised. The temporal and spatial characteristics of communication are presented for the full collection of SPLASH-2 and Parsec benchmarks. The results aim to support the design of future communication systems for CMPs, encompassing coherence protocols, network-on-chip and thread mapping.

3.1 Introduction

The communication patterns exhibited by a multithreaded benchmark are determined by a number of factors. The programming, machine and parallelisation models as well as the application algorithm all play a significant role in defining the nature of thread-to-thread communication. By using an idealised architecture for many experiments, this work aims to abstract away many of these factors, exposing the true sharing present in the algorithms used.

This work analyses a large number of applications running on a shared-memory, chip-multiprocessor (CMP) architecture. The applications are from the SPLASH-2 [79] and Parsec [15] benchmark suites. Of particular note is that the target machine model has evolved from a multi-node system (SPLASH-2) to a chip-multiprocessor (Parsec). As described by Bienia et al. [14], core-to-core communication is considerably faster on a CMP than in a multi-node system and this shift in machine model allows programs to be written using new parallelisation models previously untenable on a multi-node machine. New parallelisation models imply different communication patterns and this work aims to thoroughly characterise this shift.

The characterisation falls into four sections. Section 3.5.1 examines the basic read and write behaviour of the benchmarks. In Section 3.5.2, the spatial and temporal characteristics of thread to thread communication are examined. Data is presented showing how much sharing occurs between threads and at what times the transactions occur. This information can be used for thread mapping and interconnect topology design. Section 3.3 analyses the sharing patterns that are present in each benchmark. Three patterns are described: read-only, producer-consumer, and migratory. These patterns influence both caching policy and coherence protocol design. Finally, Section 3.5.4 explores the stability of the read sets for each communicating write issued.

3.2 Benchmark Background

For this study, two benchmark suites are used: SPLASH-2 [79], released in 1995 and Parsec, first released in 2008 [15] and updated in early 2009 [16].

SPLASH-2 is a mature benchmark suite containing a variety of high performance computing (HPC) and graphics applications. The dominant parallel platforms at the time of the suite's creation were multi-node systems, with processors often being housed

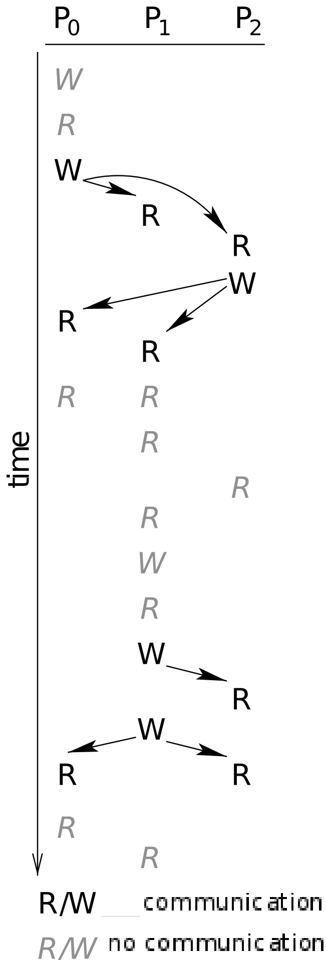


Figure 3.1: Communicating and non-communicating memory accesses.

in separate machines relying on board-to-board communication between nodes. The extremely high latency of these links required the algorithms to minimise thread-to-thread communication wherever possible. Parsec is a more recent benchmark suite, offering a wider variety of applications rather than focusing on HPC. The benchmarks are described in greater detail in Section 2.5.3.

3.3 Sharing Pattern Background

Sharing in multithreaded benchmarks can be classified in a number of ways. This section describes the terms used throughout this chapter. First, a word is described as shared if it is written to or read from by more than one processor during the execution of a benchmark. This separates the memory into shared and private regions, defining where communication could have taken place. However, not all reads and writes to such a shared region are actually used to communicate data. An application might use a refinement strategy, rewriting results until they meet a certain quality before they are communicated to other processors. As such, only the writes that produce the final value are *communicating writes*. A similar classification is possible for read operations. A read is a *communicating read*, if it reads a value that has been written by a different processor for the first time. Subsequent reads by the same processor do not communicate any new information and are an artefact of register pressure or instruction encoding (the latter is most certainly the case for x86 binaries). Figure 3.1 shows communicating and non-communicating memory accesses to an example memory location. Communicating accesses are shown in black, and non-communicating accesses are shown in grey.

The way in which shared words are accessed can be used to further categorise the memory locations. The number and ordering of reads and writes can indicate a certain sharing pattern. This chapter examines three such patterns: read-only, migratory, producer-consumer [12; 77].

Read-only A word is declared read-only if during the entire execution of a program it is written to either zero or one times, and is subsequently read by at least one processor that is not the writer. In addition, no read access is allowed before the single write access. An example of a read-only relationship is shown in Figure 3.2. Read-only sharing is

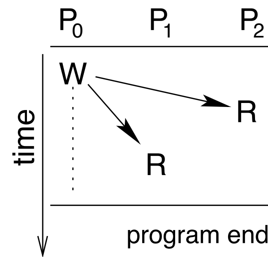


Figure 3.2: A read-only sharing memory access pattern.

most commonly observed when an input file is read into a program and the content is then consumed by several of the threads in the parallel phase of execution. In this pattern, each data word may be read several times by a variety of different processors but is never over-written once first read. Therefore any intermediate values used in further computation must be stored elsewhere. A consequence of such a pattern is that these words do not strictly require any coherence support.

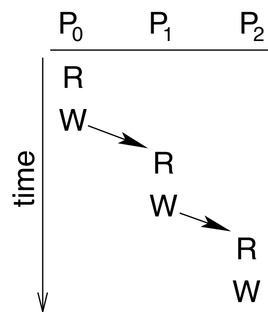


Figure 3.3: A migratory sharing memory access pattern.

Migratory This sharing pattern is found when, inside an atomic region, a shared data structure is repeatedly accessed and modified by different processors. This pattern is chracterised by a read to a newly produced data value followed by a write, without an interrupting read or write from another processor.

Migratory sharing is common in shared memory benchmarks and predictability is also high, with regions exhibiting migratory behaviour often doing so for the rest of a

benchmark's execution. Migratory sharing is of interest as it behaves sub-optimally on MESI protocols [72]. Figure 3.3 shows the first read from P1 will return with *shared* permissions, only to immediately require an upgrade for the write to *modified* state, requiring additional coherence traffic for each migration.

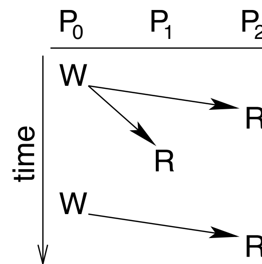


Figure 3.4: A producer-consumer memory access pattern.

Producer-Consumer This sharing pattern can be defined in a number of ways. All require a persistent relationship between sets of writing and reading processors for a given memory location. In the strictest definition, a location is only marked as exhibiting producer-consumer behaviour if each write comes from a single processor, and is always followed, before the next write, by a load from the consuming processor. The experiments, presented in Section 6.3, show that this pattern of accesses is extremely unlikely to occur multiple times without interruption. Furthermore, the producer does not remain constant. For this reason I have relaxed the definition to allow any number of writers. In this scheme, the strength of the relationship is the probability that, for each communicating write to a memory address, a communicating read will follow from a given processor. In this chapter, words are reported as exhibiting producer/consumer sharing if there is a greater than 50% probability that a specific reader will consume each write to a given location.

In addition to analysing the producer/consumer pattern directly, the stability of the reading set of processors of shared memory locations is measured. The read set for a

memory location is considered stable when for each processor, it is known with high confidence whether that processor will consume or not consume a produced value. The read set is unstable if it is not known if a processor consumes or does not consume a produced value.

Figure 3.4 shows a memory location exhibiting producer-consumer characteristics. Processor P0 acts as the producer, while P1 and P2 act as consumers. In this example, P2 is a stable consumer (since it consumes every produced value) and P1 is an unstable consumer (since it consumes 50% of the produced values). Thus, the stability of read set for this memory location is 50%, i.e. 1 in 2 processors.

This sharing pattern is important as it behaves sub-optimally under a widely used MESI cache coherence protocol [65]. The producing processor's permissions will oscillate between *modified* and *shared*, with the consumer switching from *shared* to *invalid*. In a distributed directory protocol, this would generate a large volume of messages both to and from the directory, which may be physically remote to the processing node.

Closely related to the subject of sharing patterns in parallel programs is that of invalidation patterns. This is covered in detail by work published by Gupta et al. [37].

3.4 Evaluation Setup

Simulated Architecture This work uses Virtutech's Simics simulator [57] to generate functionally correct memory accesses traces for a 32 processor x86 system running Linux 2.6.15, with a default OS configuration. Using a full Linux operating system allows a wide variety of unmodified benchmarks to run with full library support. Each processor has a single in-order pipeline, similar to the cores found in Intel's Larrabee CMP [70]. However, to maintain high simulation speed, no further pipeline details are modelled, leaving each core with a fixed throughput of 1 instruction per cycle. A cache

hierarchy of private L1s and a large shared L2 is attached to provide timing information in the traces. The private caches are kept coherent using a MESI protocol across a zero cycle, infinite bandwidth crossbar. The details are summarised in Table 3.1.

Core Count	32
ISA	x86
Pipeline	In-order, fixed CPI = 1
L1 Cache	32kB, 64B lines, 4-way associative, hit latency 1 cycles
L2 Cache	8MB, 64B lines, 32-way associative, hit latency 10 cycles
Main Memory	Latency 400 cycles
Interconnect	0 cycle, infinite bandwidth crossbar
OS	Linux 2.6.15, default configuration

Table 3.1: Simulated system parameters

Trace Generation All the experiments use memory access traces generated by a modified version of the tracer module provided by Virtutech. Chris Fensch extended the module to determine which thread is currently executed by each core, providing additional information for benchmarks that spawn a large number of threads. To retrieve this data, the tracer reads the `tr` register and follows the pointer it contains to the appropriate entry in the thread table of the Linux kernel, tagging each memory access with both the thread number and processor on which the operating system executed it. The output was optimised to reduce the size of the traces generated, but the larger files are still over 100GB.

To prevent thread migration, the OS is configured to tie threads to a specific processor. This was performed for all SPLASH-2 programs and the Parsec programs *blackscholes*, *canneal*, *fluidanimate*, *streamcluster* and *swaptions*. It was not possible to do so for other programs in the Parsec benchmark suite, as they either create more threads than processors or the threads are created in a non-trivial way.

For SPLASH-2, the simulations were run using the recommended input size for all benchmarks. For Parsec, the runs use the *simmedium* input size, keeping both simulation time and the resulting traces manageable while still accurately reflecting benchmark behaviour. As both the trace generation and replay used functional simulators, the results were deterministic for a given input set, and each simulation was only run a single time. Many of the benchmarks analysed use non-trivial input files, and exploring sensitivity to changes in these files fell beyond the scope of this work.

Communication Characterisation Consumers of a value written to memory are tracked at word-level granularity in order to identify thread-to-thread communication. This analysis is done purely at an address level, and does not take into consideration any write-back or coherence effects. On the consuming side, an infinite cache is assumed; a value that has been consumed once will always be directly accessible by the consuming node. No record is kept of any consumptions by the producing node. Furthermore, all communication that resulted from values produced during the initialisation phase is discarded, hence only measuring the communication during the parallel phase of the execution. Carrying out the analysis in this way provides a lower bound on the amount of communication that must take place, regardless of interconnect or coherence protocol design. Results from such experiments provide a useful specification for the development of on-chip communication systems.

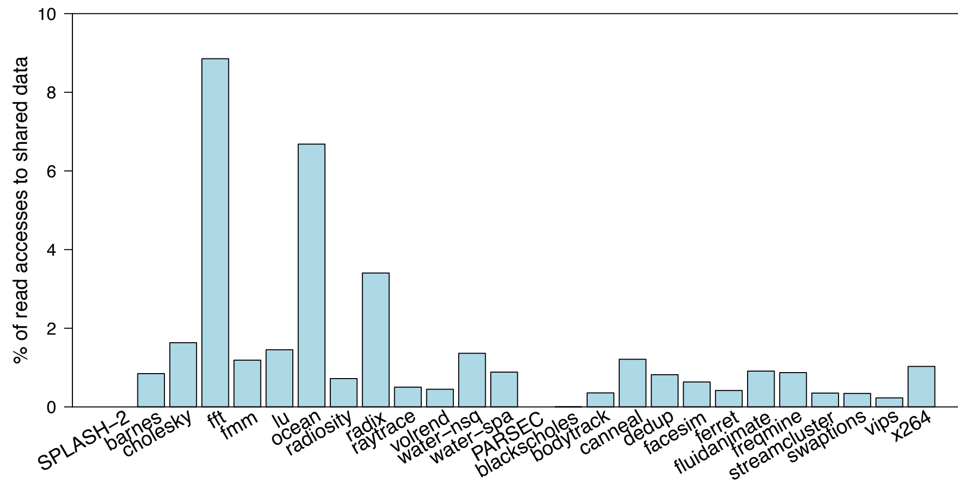
3.5 Experimental Results

This section presents the results of the communication analysis. Section 3.5.1 establishes general properties of memory accesses to shared memory locations. Section 3.5.2 investigates communication patterns, analysing which processors communicate with

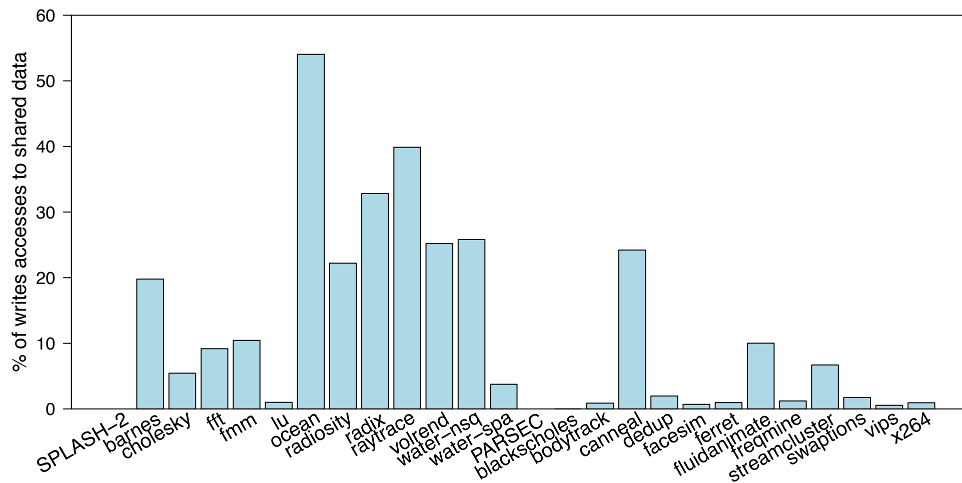
each other. Section 3.5.3 classifies the observed communication into three sharing patterns: read-only, migratory and producer-consumer. Finally, Section 3.5.4 examines how stable, and therefore predictable, the read sets of communicating write instructions are.

3.5.1 Communicating Accesses

Not all accesses to shared memory locations are used to communicate new data. Values may be re-read from memory due to lack of space in the register file or values may be refined for several iterations before being communicated. Due to the focus on communication, this analysis first identifies the number of accesses to the shared address space that communicate data, as described in Section 3.3. Figure 3.5 shows the percentage of reads and writes to shared memory locations that communicate data. On average only 1.5% of reads communicate data. However, this might be partially an artefact of simulating an x86 machine for these experiments. Due to the instruction encoding and lack of programmer visible registers on x86, it is common that almost every instruction reads from memory. Parsec benchmarks have significantly fewer communicating writes (4.2% on average) than SPLASH-2 applications (20.8% on average). This suggests a refinement of values before they are communicated. The following sections use the number of communicating accesses as the basis for many normalisations. Figure 3.6 shows the communication to computation ratio. The results show that expressing this ratio using communicating read (Figure 3.6a) or writes (Figure 3.6b), does change the absolute figures but not the general trend. An exception to this is *water-spatial*, which looks like an average communication-intensive benchmark based on the number of instructions per communicating read, but computation bound based on the number of instructions per communicating write.

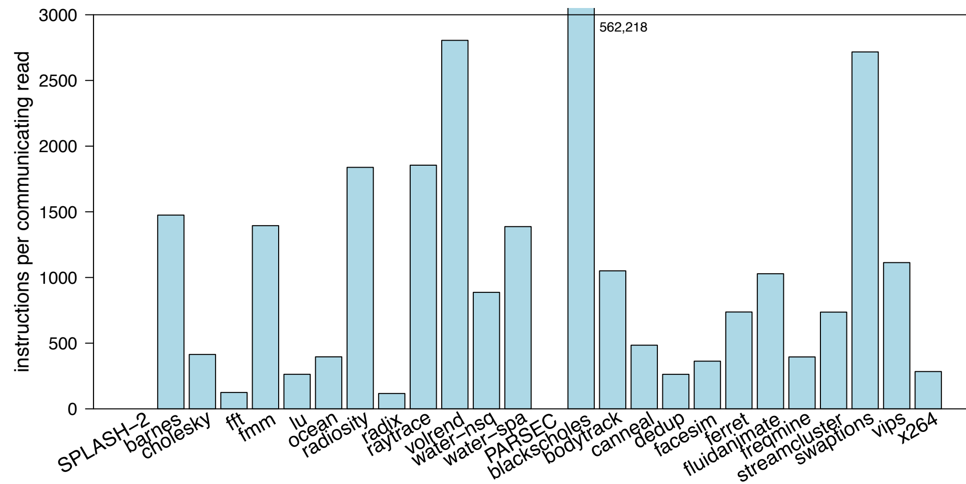


(a) Read Accesses

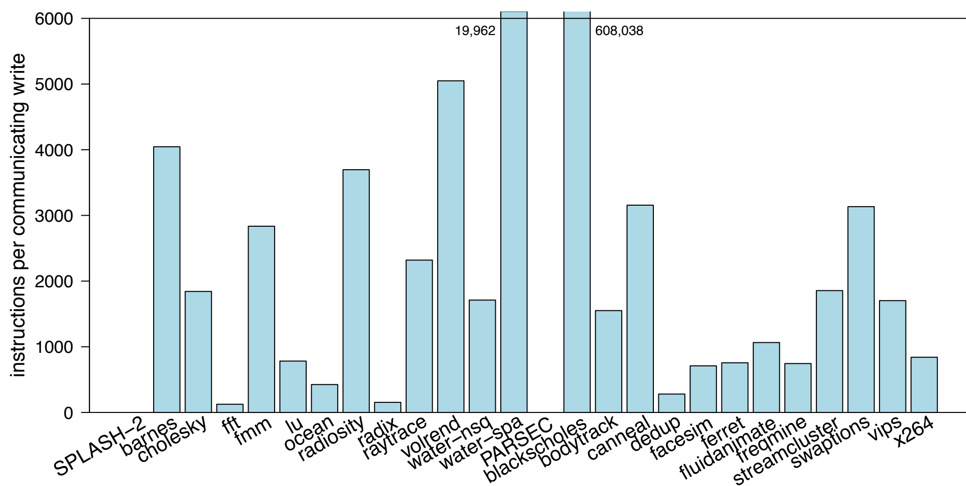


(b) Write Accesses

Figure 3.5: Fraction of read and write accesses to shared memory locations that communicate data. A read is considered communicating when it reads a value that has been produced by another processor and has not been read before. A write is considered communicating when it produces a value that is read by a different processor.



(a) Instructions per communicating read access.



(b) Instructions per communicating write access.

Figure 3.6: Instructions per communicating read and write accesses.

3.5.2 Communication Patterns

Figures 3.7 and 3.8 show the observed spatial communication patterns for the evaluated applications. Figure 3.12 shows this behaviour over time for four representative benchmarks. All plots are normalised to the maximum core-to-core communication relationship observed in that particular program. No columns or rows in the graphs have been swapped. The processors or threads appear in the order numbered by the operating system.

Spatial Behaviour SPLASH-2 programs exhibit a diverse selection of communication patterns. *Cholesky*, *lu*, *radix*, *ocean* and *water-spatial* have highly structured communication patterns that are not observed elsewhere in the benchmark selection. Second, many programs exhibit very strong communication between neighbouring processors. For example, *barnes* and *fmm* show increased neighbour communication with *blackscholes* and *streamcluster* also showing similar patterns. *Fluidanimate* exhibits a comparable trend, though each core does not communicate to its nearest neighbours but rather the 4th neighbour to either side. Both benchmark suites include a program that shows strong all-to-all communication, *fft* in SPLASH-2 and *canneal* in Parsec. Parsec contains many applications that show less uniform, but still random traffic (*dedup*, *swaptions*, *vips* and *x264*). Only two programs in SPLASH-2 show this kind of behaviour (*radiosity* and *raytrace*). A further category of programs show no recognizable pattern, but show strong communication between a few cores with almost no communication between the rest (*water-nsquared*, *bodytrack*, *facesim*, *ferret* and *freqmine*).

From a communication pattern perspective, SPLASH-2 shows more variation than Parsec. In addition, the structured patterns in SPLASH-2 often involve a high radix communication with one core communicating with 10 to 16 other cores. Parsec on the

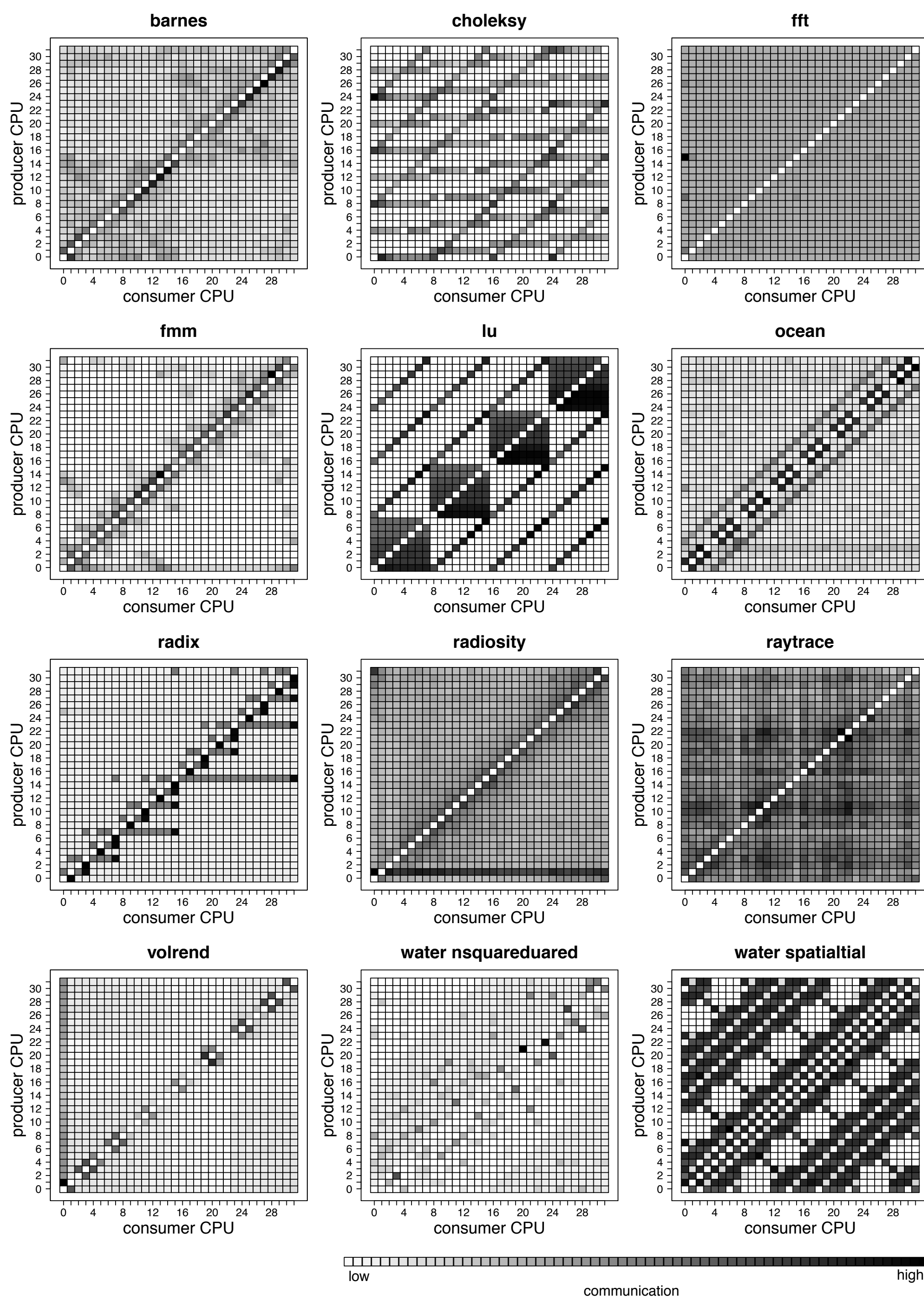


Figure 3.7: Communication between different cores during the entire parallel phase of the program for the SPLASH-2 benchmark suite, normalised per application.

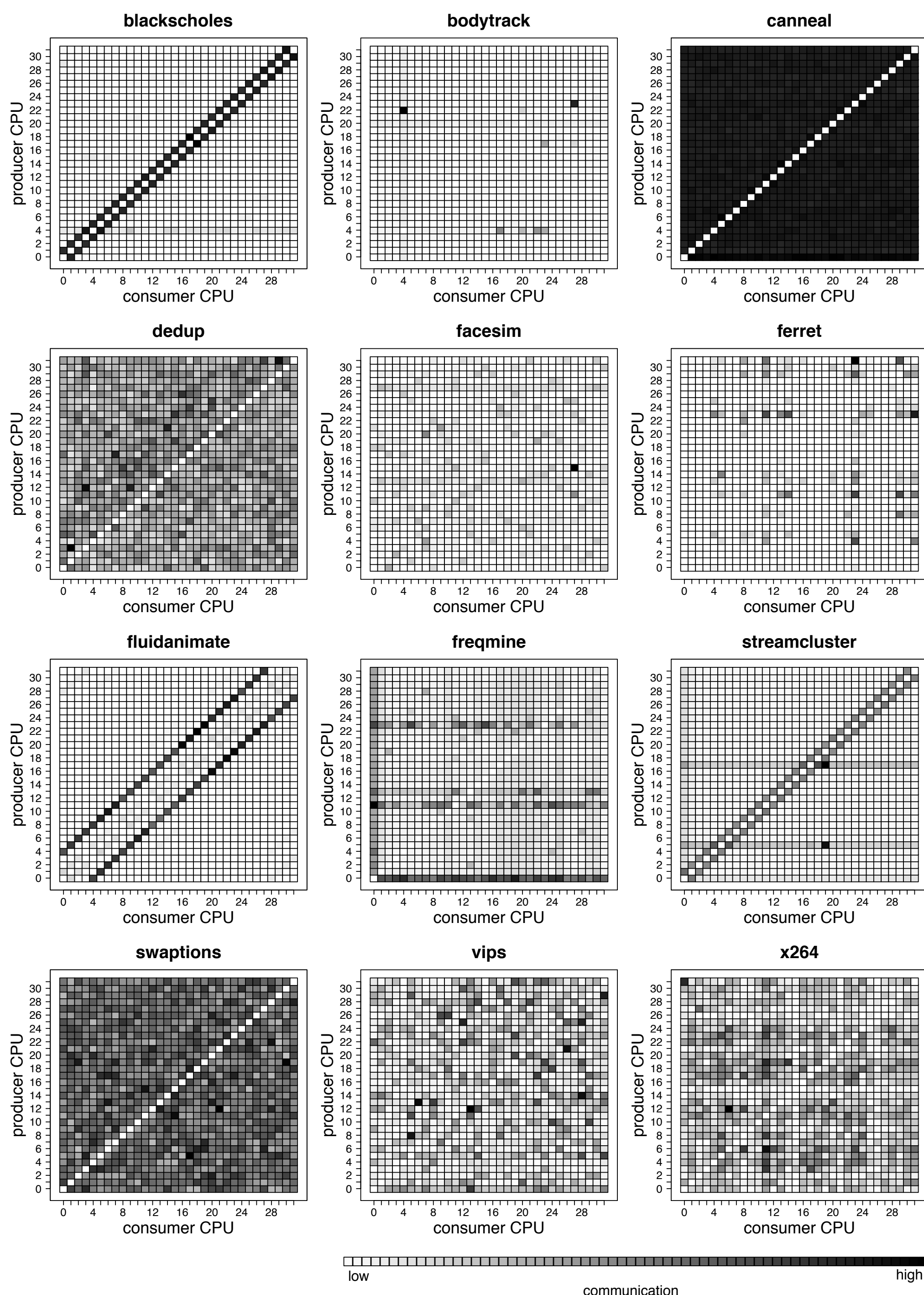


Figure 3.8: Communication between different cores during the entire parallel phase of the program for the Parsec benchmark suite, normalised per application.

other hand is dominated by either low radix or unstructured communications. All of these spatial patterns present interesting challenges for communication system design.

Cached behaviour The results for communication patterns show strong locality between near-by cores. This is a promising finding for improving communication between nodes in chip-multiprocessors. However, in order for the locality to be exploited it is necessary for it to exhibit sufficient temporal locality to be encapsulated in reasonably sized private caches. In other words, the produced data must still be locally cached at the time the consumer core generates its load request.

To evaluate this scenario, the experiment was re-run with the cache model attached. These are the only experiments in this chapter that use the cache model. Figures 3.9 and 3.10 show the results for SPLASH-2 and Parsec respectively. In some cases these results differ greatly from the infinite-sized cache results shown in Figures 3.7 and 3.8. These differences are caused by the exclusion of communication events that can be considered "uncacheable" for the given resource constraints.

Two patterns emerge. The first is that the finite-sized caches capture the majority of communication events in the SPLASH-2 benchmarks. This is illustrated by the similar patterns found in Figures 3.7 and 3.9. The second is that some Parsec benchmarks exhibit dramatically different communication patterns when run on finite-sized caches. In particular *blackscholes* and *bodytrack* show marked differences. These differences are caused by the removal of uncacheable events from the results, leading to a different normalisation. In the case of *bodytrack*, a huge number of communicating accesses between CPUs 22 and 4, and between CPUs 23 and 27 cause the normalisation to mask the underlying communication. This effect has implications for the design of real systems; the most important patterns are those that present themselves when run on realistic caches.

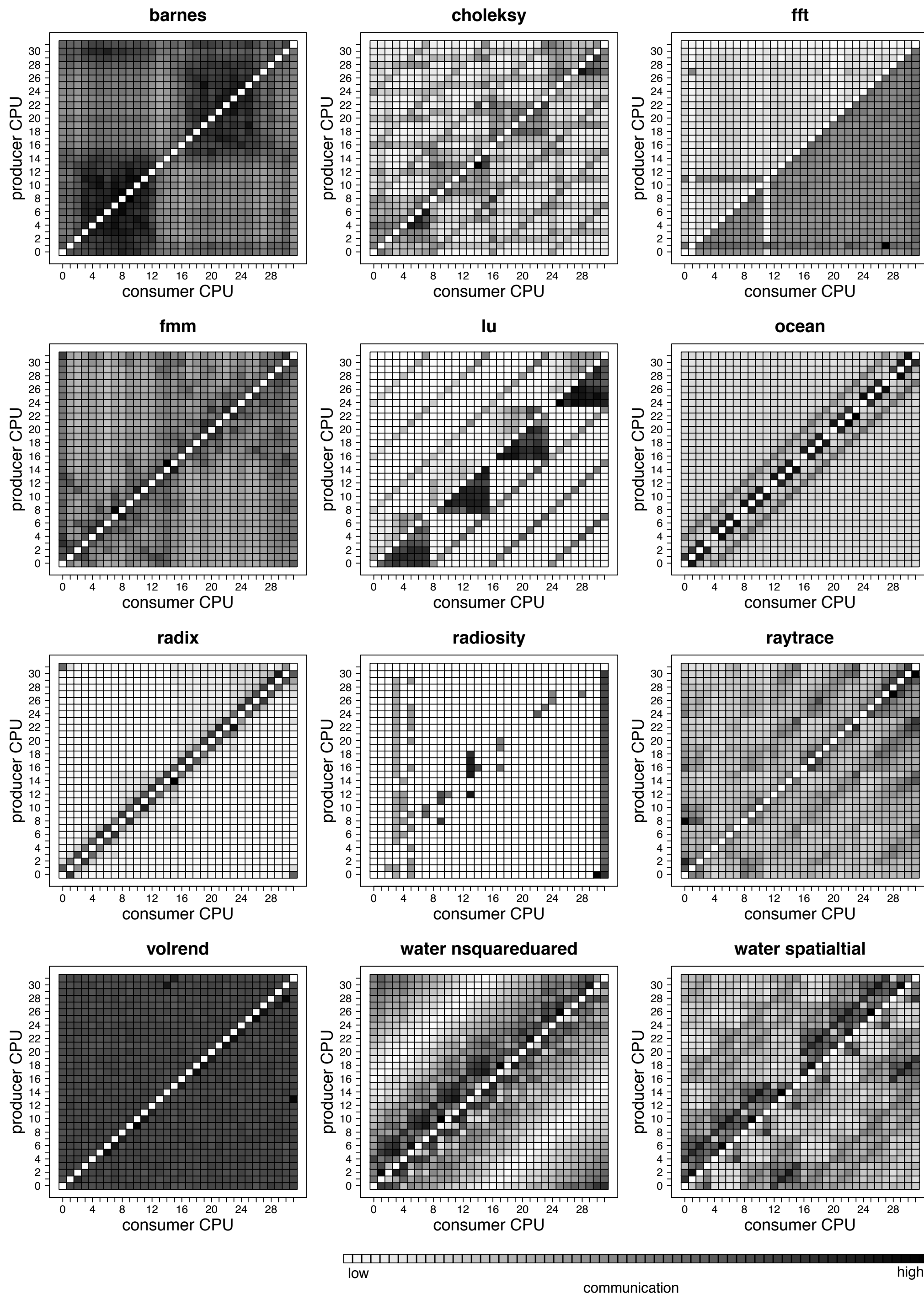


Figure 3.9: Communication between private caches of a 32 processor system running the SPLASH-2 benchmarks, normalised per application.

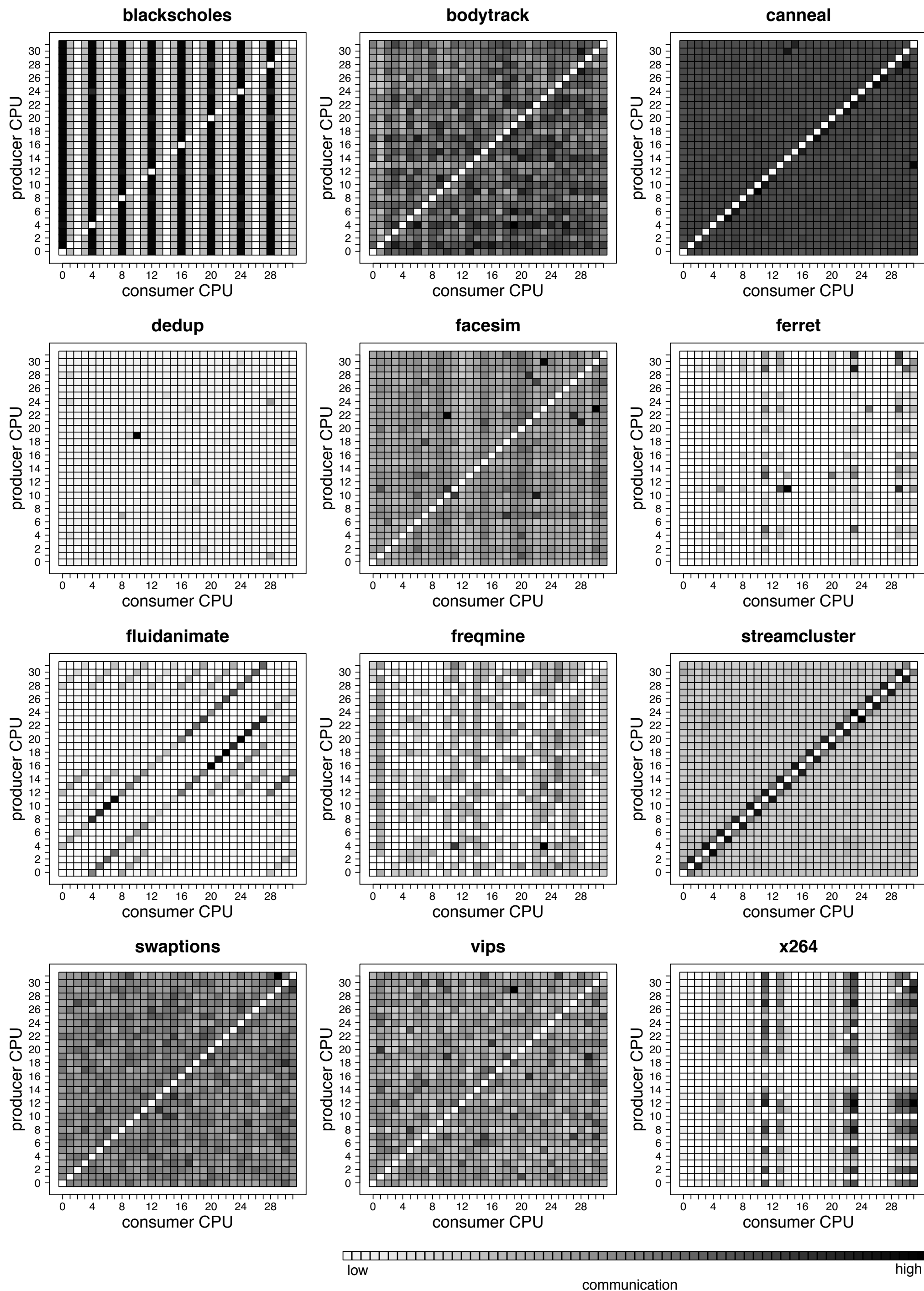


Figure 3.10: Communication between private caches of a 32 processor system running the Parsec benchmarks, normalised per application.

Ultimately, these results show that even systems with modest caches exhibit considerable locality across private caches.

Thread-Level Analysis Unlike SPLASH-2, some Parsec benchmarks dynamically generate threads during the parallel execution phase. Due to this, certain communication patterns between threads can be hidden by thread creation, mapping and migration. To eliminate this interference and expose true sharing patterns, communication is tracked based on the thread ID for programs that showed unstructured communication patterns. Figure 3.11 shows the results, again using infinite sized caches, for *dedup*, *ferret* and *x264*. In all cases distinct communication patterns become visible that were previously hidden.

Dedup generates three classes of threads that exhibit different behaviour: the first group (threads 33 to 64) produces data, which is consumed by the second group (threads 1 to 32). However, only 8 threads in this group produce any significant amount of data that is later consumed. The threads in the second group collaborate in groups of 4 threads to produce data for the third group (65 to 96). The threads in the third group show random communication among themselves.

Ferret spawns the largest number of threads of all Parsec programs (133 threads). The first 32 threads show very strong nearest neighbour communication, while the remaining threads show very limited communication. This suggests that the mapping of the first threads is of much greater importance than the higher indexed threads.

X264's thread-based communication pattern shows that half of the spawned threads exhibit little communication. For the other half, a strong communication with 5 other threads can be identified, likely due to the sharing of frames in the compression algorithm.

The strength and regularity of the sharing exposed by performing thread based ana-

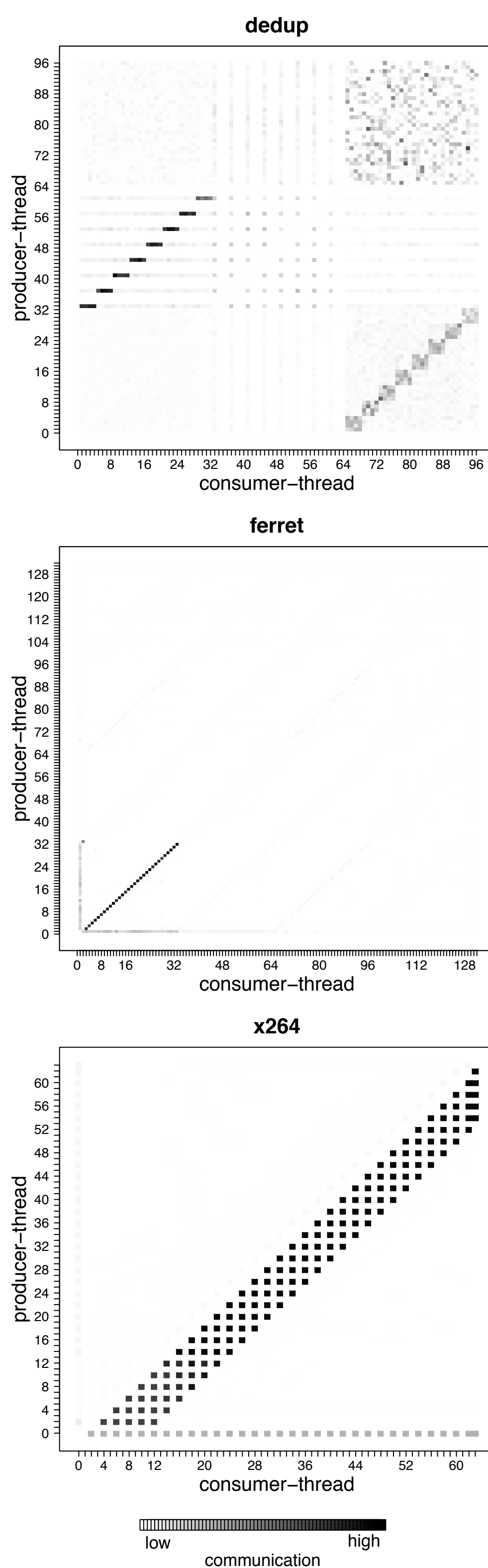


Figure 3.11: Communication between different threads during the entire parallel phase of the benchmark, normalised per application.

lysis has implications for thread mapping in Parsec benchmarks. A more intelligent spawning and mapping may well lead to clearer locality being observed in the processor level results.

Temporal Behaviour The results presented so far focus on the spatial behaviour of the benchmarks. However, the temporal behaviour of the communication is also of upmost importance when considering interconnect design.

Figure 3.12 shows the temporal communication behaviour of a single processor for four programs. With the exception of *canneal*, it is possible to identify patterns in the communication behaviour over time. Even if a core communicates with every other core during the program execution, it is not necessarily the case that every core receives all communications. For example, processor 2 in *barnes* only communicates with all other processors during very short phases in the program's execution. For the first quarter, there is some light traffic directed to cores 16 to 31. After a short period in a synchronisation phase which results in communication to all other cores, the focus of communication shifts to cores 0 to 15. During this phase, there is also a period of heavy communication with processor 1, for approximately 10% of the total execution time. A similar behaviour can be seen in *bodytrack*: for the majority of the parallel phase there is little communication between nodes. During two separate phases that cover approximately 30% of the execution time, there is all to all communication. This communication is mostly light, but during the first phase there are heavy bursts targeted at processors 4, 6, 8, 13, 22, 23, and during the second phase targeted at processors 0, 3, 6, 7, 10–14, 22 and 24. Another interesting communication pattern is seen in *streamcluster*. While there is some random, light communication to all other nodes, the results show that, for 15% of the execution time, there is heavy communication present to processor 18 and 20. For the remaining time, light traffic is observed.

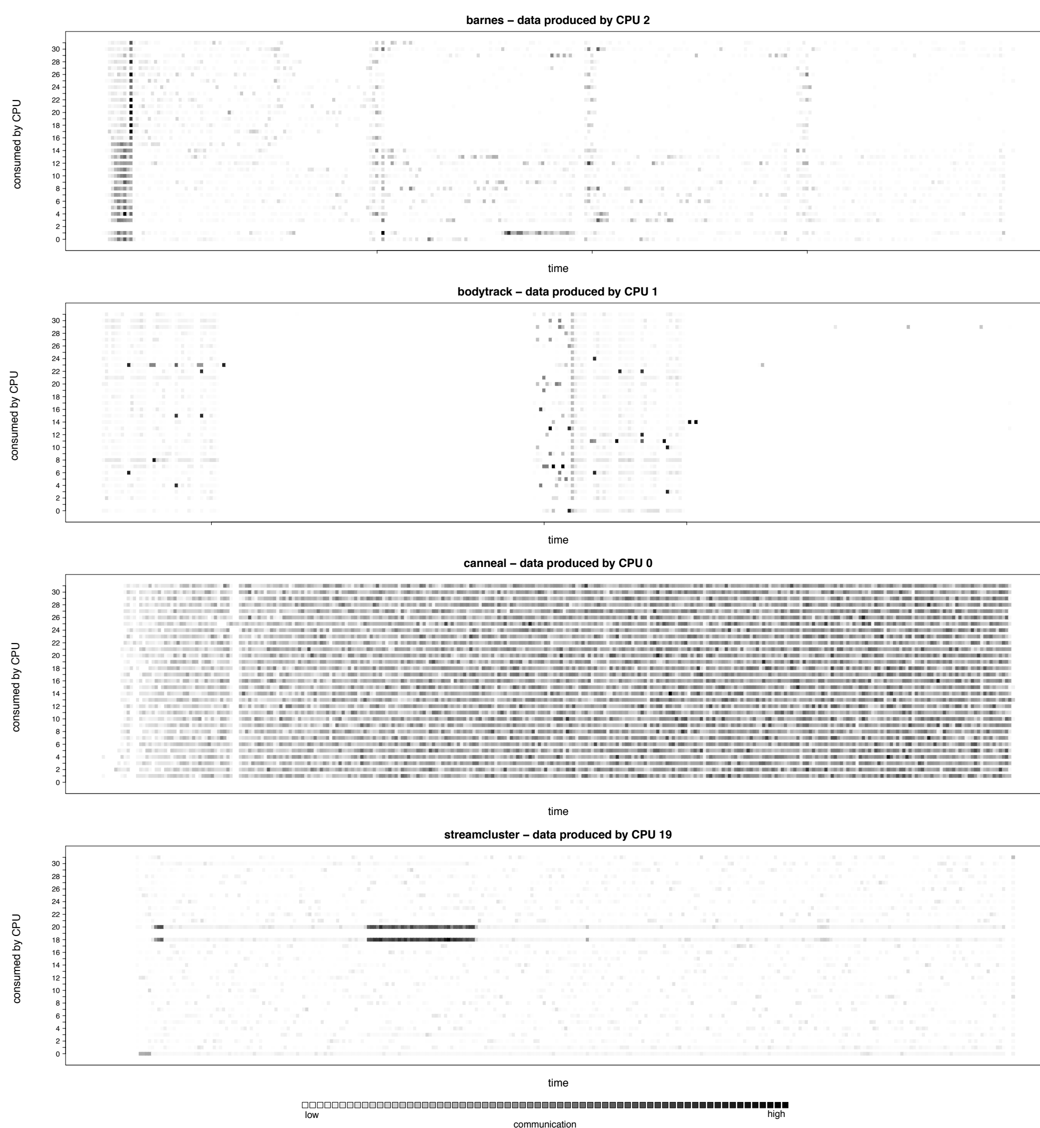


Figure 3.12: Communication changes over time for a selection of processors and applications, normalised per application.

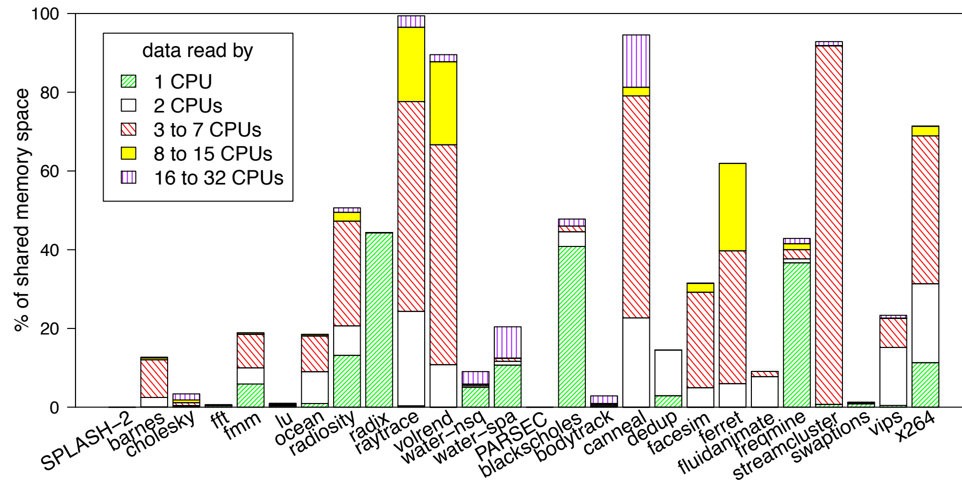
3.5.3 Sharing Patterns

Figures 3.13a, 3.14a and 3.15a show the proportion of the shared memory space exhibiting each of the three sharing patterns described in Section 3.3. While a location can only be read-only shared, it can under certain conditions participate in both a producer/consumer and migratory behaviour. The results show how many different nodes access the memory location. For producer/consumer and read-only sharing, this indicates the number of different cores that consume the value. For migratory sharing, it shows the number of different processors that participate in the migratory pattern over the entire parallel phase.

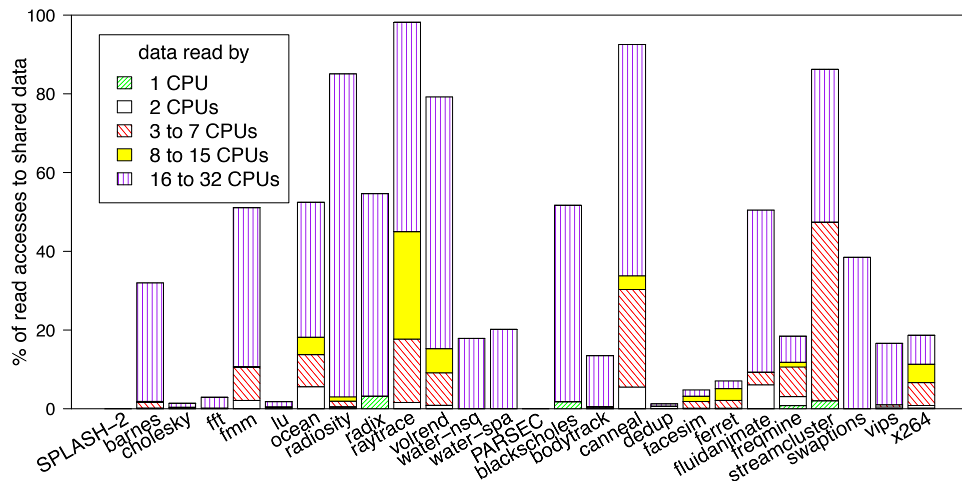
For 9 out of 24 programs the sharing characterisation scheme covers almost all shared memory locations. For another 7 programs, 50% or more of shared memory locations fit into the classifications. The remaining programs do not exhibit any recognised sharing pattern. This is best described as a multiple producer/multiple consumer. Finally, the results show that, with the exception of *water-spatial*, *water-nsquared* and *canneal*, few memory locations are involved in a communication involving more than 8 cores.

Read-Only Sharing Figure 3.13a shows the percentage of the shared memory space that is used for read-only sharing. It is further divided by the number of different cores that read a word from this space. *raytrace*, *volrend*, *canneal*, *streamcluster* and *x264* use almost all of the shared address space in a read-only manner and to a lesser extent *radix* and *ferret*. While there is some data that is being read by 16 or more processors, most sharing is performed between up to 7 processors.

Figure 3.13b presents a quantitative analysis of read accesses to shared data. Most applications that use their shared address space in a predominantly read-only manner also direct most shared reads to these regions. The exceptions are *ferret* and *x264*,

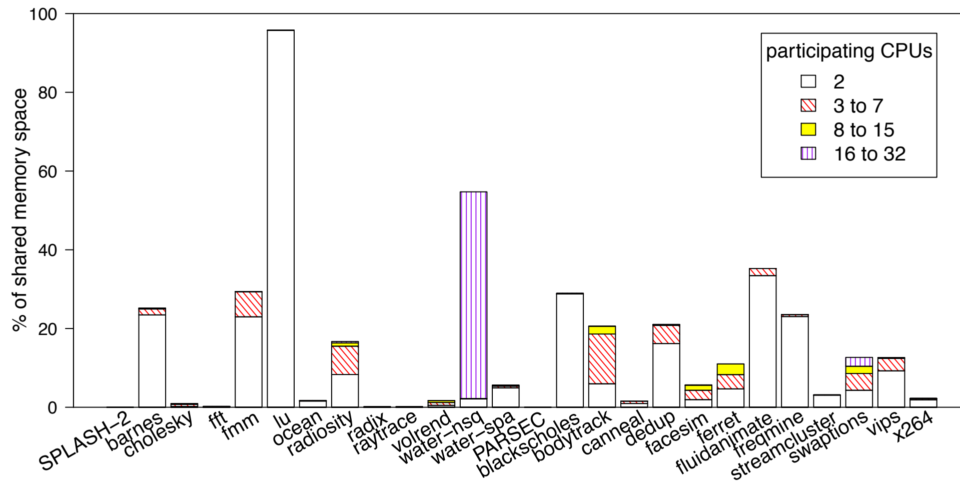


(a) Spatial

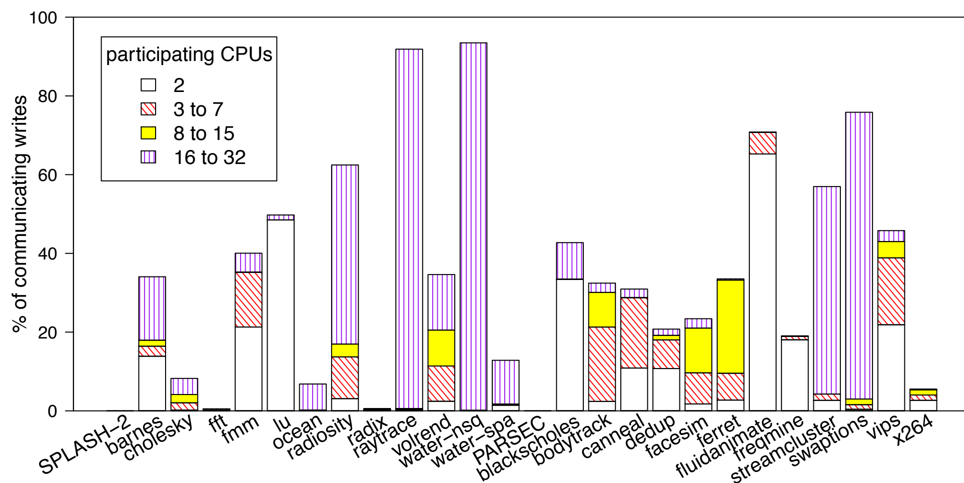


(b) Quantitative

Figure 3.13: Analysis of the read-only sharing pattern. The spatial analysis shows the percentage of the shared address space that is used according to the read-only sharing pattern. The quantitative analysis shows the percentage of reads to shared address space that access a location that had been classified as read-only. For both analyses, the number of processors the line is read by is used to classify each access (Read-only locations with only one reading processor, are written by a different processor).

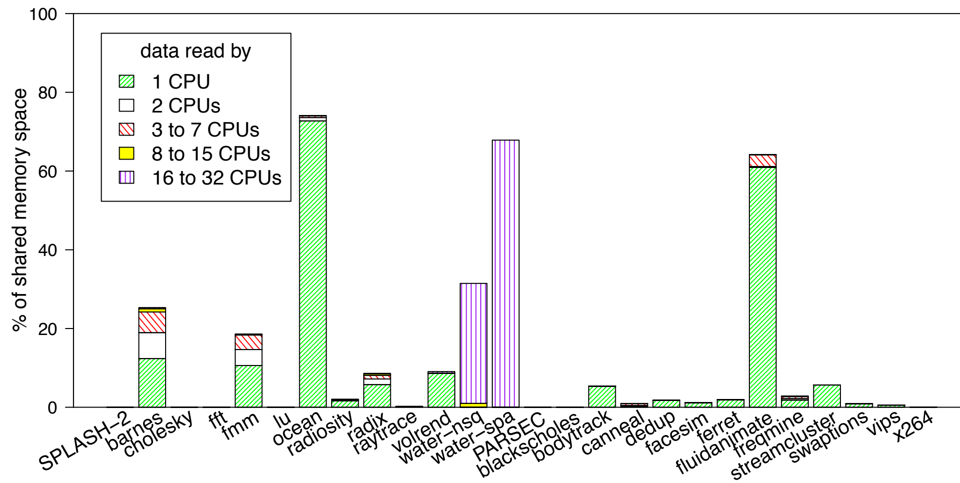


(a) Spatial

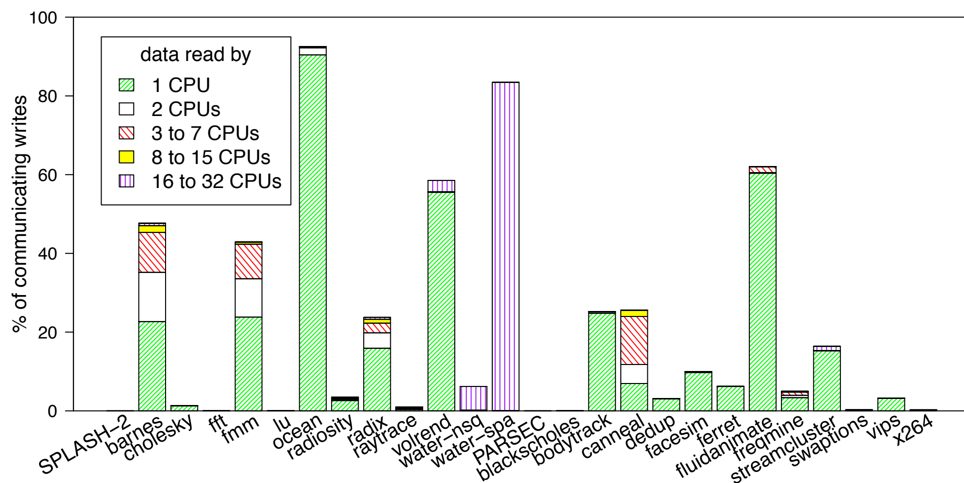


(b) Quantitative

Figure 3.14: Analysis of the migratory sharing pattern. The spatial analysis shows the percentage of the shared address space that is used according to the migratory sharing pattern. The quantitative analysis shows the percentage of communicating writes that access a location that had been classified as migratory. For both analyses, the number of processors participating in the migratory patter is used to classify the write accesses.



(a) Spatial



(b) Quantitative

Figure 3.15: Analysis of the producer-consumer sharing pattern. The spatial analysis shows the percentage of the shared address space that is used according to the producer-consumer sharing pattern. The quantitative analysis shows the percentage of communicating writes that access a location that had been classified as producer consumer. In both experiments, accesses are classified by how many processors consume the data.

which use 61% and 71% of their shared memory space in a read-only way, but only 7% and 19% of their read accesses read this data. Several benchmarks (*fmm*, *ocean* and *fluidanimate*), which do not use a significant portion of their address space for read-only data, direct 40% to 50% of their shared reads to these regions.

Migratory Sharing Figure 3.14a shows the percentage of shared memory locations that participate in migratory patterns. It is further divided by the number of different nodes that participate in this pattern. Only five SPLASH-2 benchmarks (*barnes*, *fmm*, *lu* and *water-nsquared*) use a noticeable fraction of their shared memory space for migratory data. In Parsec, all benchmarks apart from *canneal*, *streamcluster* and *x264*, use a significant amount of the shared memory space for migratory communication. An analysis of how many cores use a particular memory location for a migratory sharing pattern shows that most migratory locations are only being used by 2 cores. A few locations are used by up to 7 cores. The only exceptions to this are *water-nsquared* and *swaptions*. In *water-nsquared*, almost all migratory locations are shared between all processors. In *swaptions*, about two thirds of the migratory address space is used by more than seven cores.

Figure 3.14b shows the percentage of communicating writes that participate in migratory sharing patterns. All applications exhibit some extent of migratory behaviour. The results display a full range values, suggesting optimisation of migratory patterns is important, but will never yield universal improvements.

The Parsec benchmark suite exhibits more migratory sharing pattern than SPLASH-2. Migratory patterns are easier to support in a CMP environment than in a multi-node system and it is no surprise to find them more heavily used in Parsec.

Producer-Consumer Sharing Figure 3.15a shows the percentage of shared memory locations that participate in a stable producer-consumer relationship as defined in Section 3.3. The results are further divided by the number of different cores that consume the word produced. The first observation is the almost complete absence of stable producer/consumer memory locations in Parsec, with the exception of *fluidanimate*. Second, only five SPLASH-2 applications use a significant amount of shared memory space for producer-consumer patterns: *barnes*, *fmm*, *ocean*, *water-nsquared* and *water-spatial*. Third, there is a large variance in the number of nodes that are involved in producer consumer patterns. In *water-nsquared* and *water-spatial*, all nodes participate but for the other four applications, most data is consumed by only a single node. This suggests that using broadcast techniques in an on-chip interconnect or coherence protocol is likely to benefit *water-nsquared* and *water-spatial*, but it will be of limited use for almost all other applications.

Finally, *water-nsquared* and *water-spatial* are the only programs that exhibit a significant amount of sharing of data between more than 15 cores. The only program in the Parsec benchmark suite that shows such a high degree of sharing is *canneal*, and as shown in Figure 3.15b, even then only for read-only data.

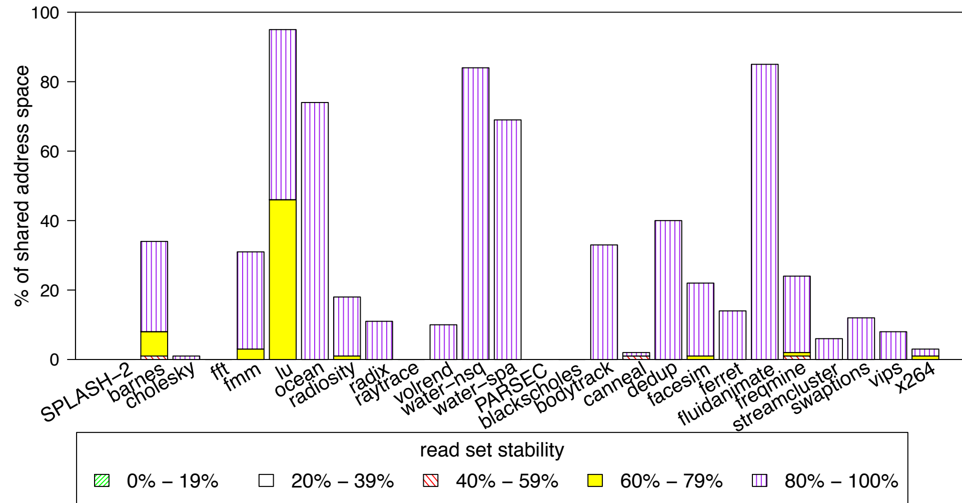
Figure 3.15b shows the percentage of communicating writes that access a location with a stable producer-consumer relationship. The main observation is that applications that use a significant fraction of the shared address space for producer-consumer communication also use a signification fraction of communicating writes in this way. The two exceptions to this observation are *volrend* and *water-nsquared*. *Volrend* only uses around 10% of the shared address space for producer-consumer communication, but more than 55% of its communicating writes. *Water-nsquared* uses around 35% of its shared address space for producer consumer communication, but only 7% of its communicating writes.

3.5.4 Read-Set Stability

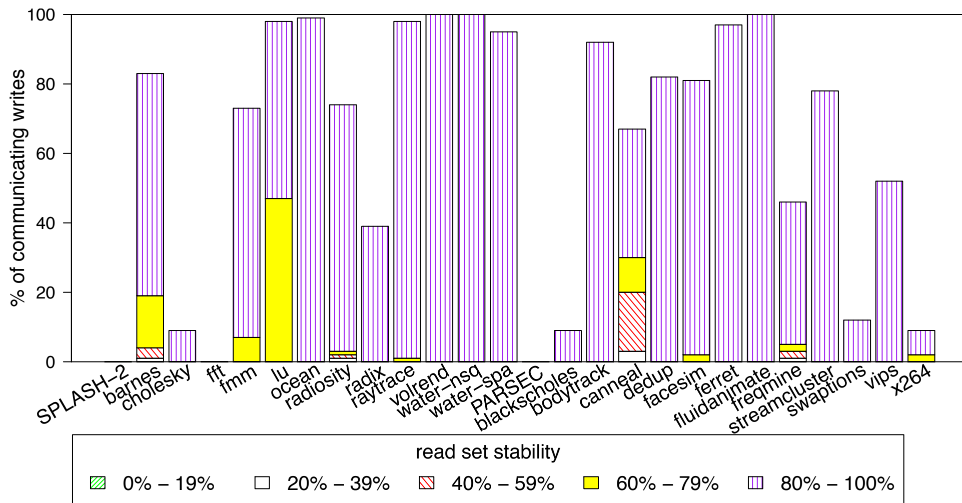
The read set is considered stable when it is known that a produced value will be consumed or not be consumed by a given processor. A processor that always consumes a produced value contributes to a stable read set. Similarly, a processor that never consumes a produced value also contributes to a stable read set. A processor that consumes only half of the produced values contributes to an unstable read set. Hence, a migratory sharing pattern will be classified as a stable read set. In order to classify a location as stable, it is necessary that at least two communicating write accesses are performed on that location.

Figures 3.16a and 3.16b show the results for the stability of the read set. In both the spatial and quantitative analyses, a significant number of locations and write accesses have a very stable read set (80% to 100%). In many cases these results correlate with the migratory sharing results in Figure 3.14. Minor differences in these results, such as when more locations are classified migratory than there are locations with a read set stability, are due to slight differences in the classification of these locations. For example, the last write in a migratory pattern does not have to be a communicating write. This means if a migratory pattern consists of only 2 writes then it is possible that it will not be considered for the read set stability analysis.

Exceptions to the correlation are *ocean*, *radix*, *volrend*, *water-spatial*, *bodytrack*, *dedup* and *ferret*. These benchmarks show a highly stable read set, which is not the result of a migratory sharing pattern. In general, stability in the read set is due to knowing that processors are not going to read a produced value. This behaviour is already exploited by current cache coherence protocols, which assume a value is not being consumed. To measure the stability of the read set it is necessary to increase the threshold for detecting a stable producer-consumer relation to the region of 70%



(a) Spatial



(b) Quantitative

Figure 3.16: Stability analysis of the read set of produced values. In order to characterise the stability of a location, it is necessary that at least two communicating writes are performed. The spatial analysis shows the percentage of shared address space with two or more communicating writes. The quantitative analysis shows the percentage of communicating writes that access a location with two or more communicating writes. For both analyses, accesses are classified according to the read set stability for the relevant memory location.

Program	Min	Max	Avg
barnes	2	5,519	2
cholesky	2	1,128	289
fft	2	446	20
fmm	2	2,141	4
lu	2	4,282	115
ocean	2	53,230	12
radiosity	2	229,744	61
radix	2	574	12
raytrace	2	130,899	28,052
volrend	2	2,335	2
water-nsq	2	954	18
water-spa	2	955	10
blackscholes	32	64	32
bodytrack	2	10,101	251
canneal	2	4,095	152
dedup	2	4,451	451
facesim	2	27,834	22
ferret	2	857	30
fluidanimate	5	2,558	11
freqmine	2	1,633	38
streamcluster	2	826,793	4,132
swaptions	2	12,914	1,684
vips	2	4,289	83
x264	2	1,085	17

Table 3.2: The minimum, maximum and average number of communicating writes per line, i.e. those shown in Figure 3.16.

to 90%. Figure 3.16b shows the results of the quantitative analysis. *Barnes*, *canneal*, *fluidanimate*, *fmm*, *ocean*, *radix*, *volrend* and *water-spatial* have a significant fraction of read set stability due to knowing which processors will consume a value.

Since a location can exhibit a stable read set with just two communicating writes, the number of communicating writes for each locations can be broken down further. Table 3.2 shows these results. Only in *barnes*, *fmm* and *volrend* have memory locations with fewer than five communicating write accesses on average. All benchmarks show a significant number of communicating writes per memory location, suggesting that it is worthwhile to exploit read set stability in communication optimisation.

3.6 Conclusions

This chapter presents a detailed analysis of the communication exhibited by the SPLASH-2 and Parsec benchmark suites. It shows that using detailed functional simulations at the thread level facilitates the characterisation of communication relationships otherwise masked by OS mapping and scheduling policies. The infrastructure provides sufficient speed to analyse the full duration of each benchmark, giving an insight into the temporal behaviour of the communication patterns. These results have an impact on a number of areas of processor design.

Thread Mapping By analysing communication at a thread level, it is possible to see that default OS-level thread mapping policies do not optimise for physical locality of shared data. Some level of manual control is possible through the use of thread-affinity masks, but this places considerable burden on the user. On current platforms, it is unlikely that a thread mapping will cause problems, but in an architecture with less uniform communication costs, this may be of increasing concern. Further research could

characterise the performance benefit of using this information in future CMP platforms, and certainly the ideas presented in later chapters of this work would benefit from such advances.

Coherence Protocols By classifying shared memory locations and accesses into read-only, migratory and producer/consumer, researchers can predict which benchmarks will benefit most from communication-aware optimisations. Existing protocol modifications, such as those presented by Cox et al. [26], targeting migratory sharing should see good improvements on the emerging workloads in the Parsec suite. Producer/consumer sharing however, is harder to find, and schemes aiming to optimise for this behaviour may need to do so at a finer temporal granularity than used here. Finally, the large amount of read-only sharing present in many of the benchmarks reminds researchers to maintain good support for this basic pattern.

On-Chip Interconnect Many of the spatial and temporal results have an impact on interconnect design for CMPs. It is evident that there is no common case communication behaviour and that the traffic is rarely constant over time, placing demands on the interconnect architecture. The locality of the spatial communication has implications for the network topology choices a designer makes, but the temporal properties must also be considered. For example, clustering compute nodes to aggregate traffic may lead to congestion in the higher traffic phases of program execution. Additionally, a number of the characteristics presented here could be combined to provide synthetic traffic patterns for router design and evaluation. The infrastructure developed for this work has already been used to investigate on-chip optical interconnects [67].

CHAPTER 4

Physical Locality

As established in Chapters 2 and 3, architects must now optimise for communication when designing chip-multiprocessors. More specifically, designers must strive for efficiency in the communication layer of new systems, due to strict power budgets at both the chip and board level.

To meet these constraints, hardware is now designed to maximise performance per Watt spent, and the only way to evaluate such metrics is through workload driven design techniques. Chapter 3 demonstrates that by thoroughly examining the behaviour of parallel programs, it is possible to find new patterns to be exploited when designing parallel systems.

Of particular interest are the results concerning locality of memory accesses in each workload. Traditionally, locality has meant the temporal or spatial proximity of loads and stores through a single processor's memory system. In this situation, spatial locality refers to the address space of a program.

However, Section 3.5.2 describes a new kind of locality found in parallel systems. Memory accesses that communicate data exhibit strong locality across adjacent processor and thread IDs. This work refers to this behaviour as *physical locality*, as when an application is mapped to a chip-multiprocessor, the data to be communicated will be in close physical proximity on the die.

The impact of this new locality is hinted at when considering a traditional cache coherence protocol, such as those presented in Section 2.3. When running parallel ap-

plications that use frequent thread-to-thread communications, data is rarely fetched in the most efficient way. For example, a node that misses in its private cache will always go first to the directory structure before being redirected to the node holding the requested line. The results in Section 3.5.2 show that this node is often close to the original requester, and therefore data is being fetched in an inefficient manner. The situation is exasperated by the high power cost of unnecessarily accessing the global interconnect to reach the potentially remote directory.

If this locality is considered during the design of the communication layer of chip-multiprocessors – from the physical network, to the coherence protocol – is it possible to increase efficiency by fetching data from the node in the nearest physical proximity?

4.1 Initial Study

Section 3.5.2 shows the layout of shared data across the processors of a parallel system and the sharing patterns that a coherence protocol could be optimised for. However, the results do not describe the ratio of accesses that can be satisfied by physically local data, or the coherence state in which the shared data is found.

To answer this question, memory access traces of all programs in the SPLASH-2 [79] and Parsec [15] benchmark suites were run on a functional simulator of a MESI cache coherence protocol. The traces were gathered at the processor level; all memory accesses were included, regardless of their hit-rate in any hypothetical cache hierarchy. The experiments used infinite sized caches and simulate a system in which any local cache miss checks all other private caches in the system for a copy of data suitable for forwarding.

Under the baseline MESI protocol there are three ways in which physically local data can be forwarded:

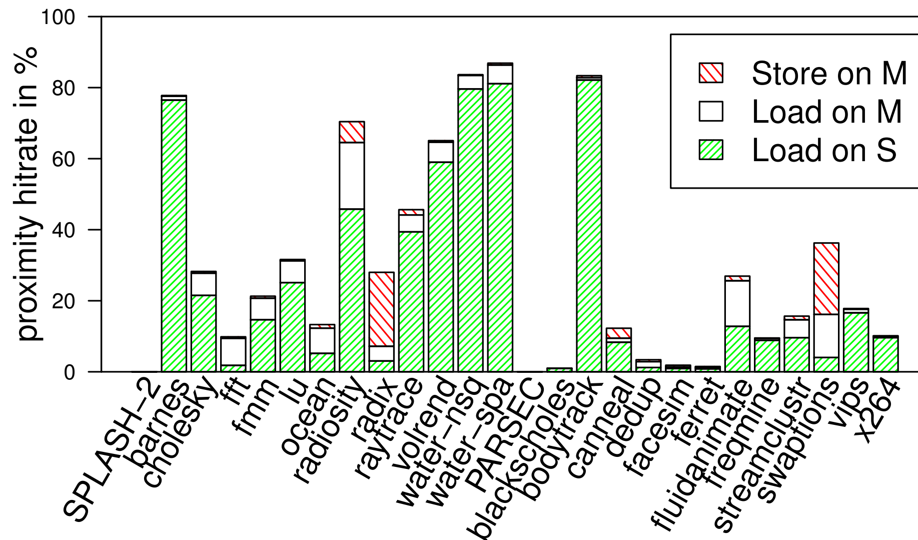


Figure 4.1: Results of the oracle study to investigate the limits of physical locality in a 32 core system. When an L1 cache miss occurs, the core checks all L1s for data that could be forwarded.

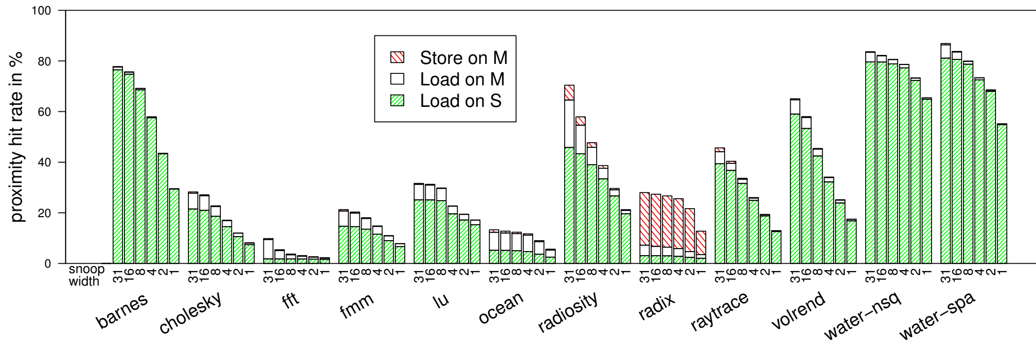
Load on S The requester performs a load operation and snoops a cache that has the data available in state **S**hared. The data can be forwarded to the requester.

Load on M The requester performs a load operation and snoops a cache that has the data available in state **E**xclusive or **M**odified. The data can be forwarded to the requester. However, in order to maintain coherence, the snooped cache can no longer write to its cache line without invalidating the requester's copy first.

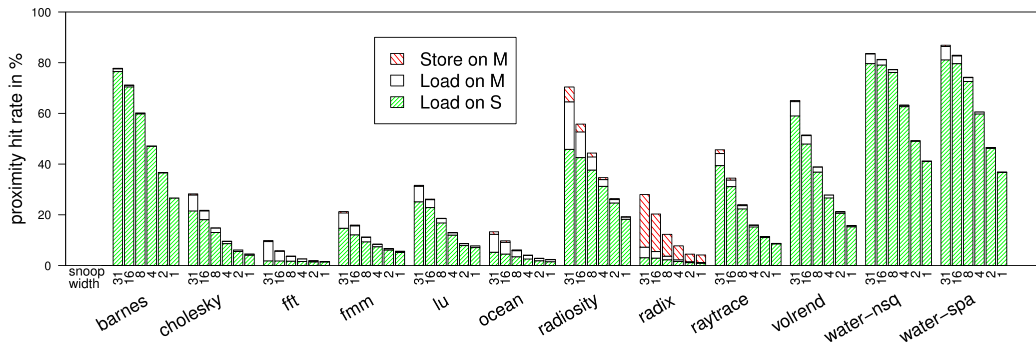
Store on M The requester performs a store operation and snoops a cache that has the data available in state **E**xclusive or **M**odified. The data and write permissions can be forwarded to the requester. However, the snooped cache can no longer read or write its cache line, without getting an up-to-date copy back first.

The sum of these three classifications gives the number of cache misses that could be satisfied by a coherence protocol exploiting physical locality.

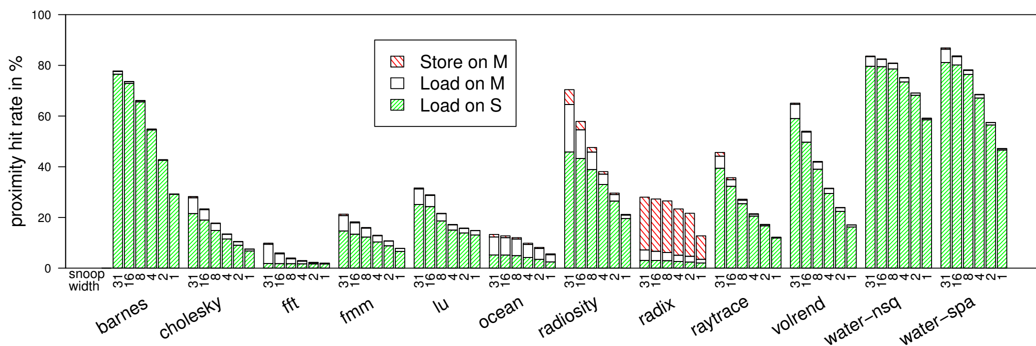
Figure 4.1 shows that all programs exhibit at least some degree of locality, with



(a) Ideal mapping

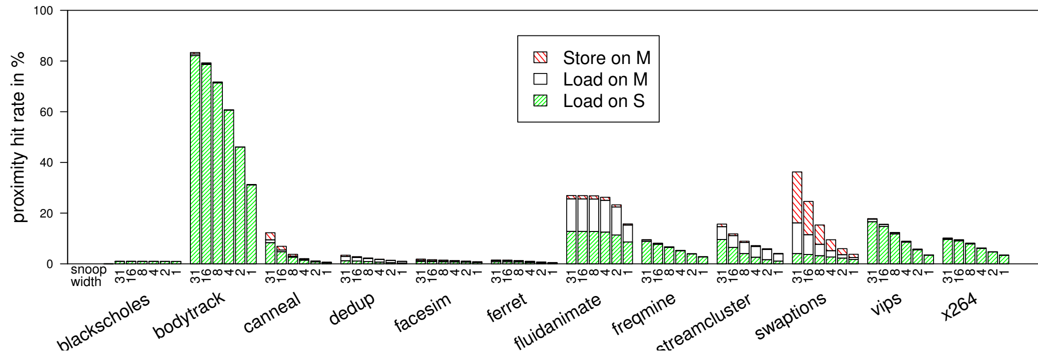


(b) Random mapping

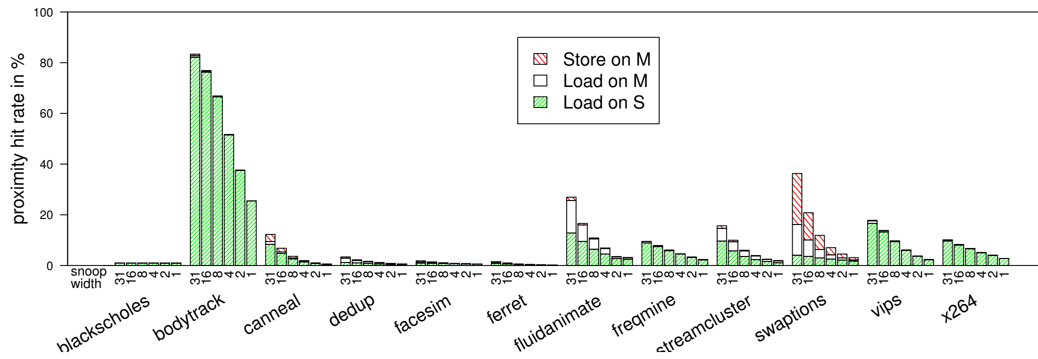


(c) H-tree mapping

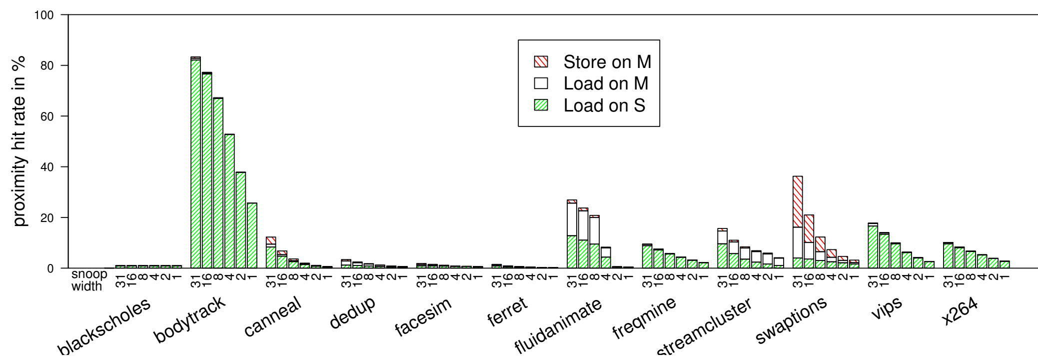
Figure 4.2: Impact on the proximity hit rate of SPLASH-2 benchmarks when the number of cores snooped is reduced. In this study, snooping is limited to n neighbours. The study evaluates three policies for neighbour selection: *Ideal*, *Random* and *H-tree*



(a) Ideal mapping



(b) Random mapping



(c) H-tree mapping

Figure 4.3: Impact on the proximity hit rate of Parsec benchmarks when the number of cores snooped is reduced. In this study, snooping is limited to n neighbours. The study evaluates three policies for neighbour selection: *Ideal*, *Random* and *H-tree*

many showing considerable potential for optimisation. For example, *barnes*, *bodytrack*, *volrend*, *water-nsquared* and *water-spatial* all exhibit hit rates between 65% and 87%. Across all benchmarks, *Load on S* and *Load on M* events cover over 95% of all hits. *Store on M* events occur infrequently, as shared data is almost always read before it is overwritten. *Radix* and *swaptions* are the only exceptions, exhibiting a significant fraction of *Store on M* events due to false sharing. Importantly however, it is necessary to check every other cache in the system after each local cache miss to achieve such hit rates.

A two phase study is used to investigate the effects of limiting the number of processors snooped on each cache miss. Using results from the experiments in which all caches are snooped, ordered lists of "preferred neighbours" are generated for each core in the system ; a preferred neighbour is a cache that is more likely to return a proximity hit when snooped. The following offline approach was used to generate the lists:

- Run all benchmarks through the simulator, configured to snoop all 31 other processors for valid data on a cache miss.
- For each processor, record a table of successful and unsuccessful snooping attempts (hit/miss events)
- When the simulation completes, for each benchmark calculate the hit-rate of snooping attempts made from each processor to every other.
- For each processor, order the table by descending hit-rate.
- This generates an application specific list of "preferred neighbours" for each processor.

The second phase of the study uses these lists to determine hit rates when snooping only the first 1, 2, 4, 8, 16 or 31 caches, as shown in Figures 4.2a and 4.3a.

8	11	29	2	22	5	0	23
16	31	12	10	14	25	1	24
28	13	27	19	3	26	30	4
15	6	9	7	20	21	18	17

Figure 4.4: An example random thread mapping used to evaluate physical locality

These initial experiments show that not only is there regular physical locality in many shared memory parallel programs, but also that a substantial number of local cache misses are to regions of data already held elsewhere in the system. The results generated from the functional cache coherence simulator show that a sizeable proportion of benchmarks can benefit from schemes optimising for physical locality of shared data. This is a significant result, as the benchmarks that have high locality can be expected to benefit greatly from mechanisms such as Proximity Coherence, presented in Chapter 5.

However, using the neighbour lists in this way does not take into consideration the topology restrictions of an actual chip-multiprocessor design. Two further scenarios were evaluated under the constraints of a 8x4 2D mesh to reveal if physical locality remains.

The first experiment was a random mapping combined with the 2D topology constraints described above. An example of such a mapping is shown in Figure 4.4, The results are shown in Figures 4.2b and 4.3b. A sizeable amount of locality remains even with a random mapping, but the hit rate at lower snoop widths is drastically reduced. Unfortunately, these low snoop width scenarios are those most likely to be suitable for exploitation in hardware. For this reason it is desirable to find a 2D mapping that captures a greater percentage of the locality found in the ideal mapping.

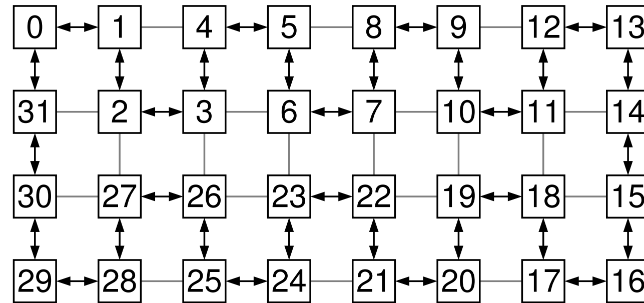


Figure 4.5: An H-tree thread mapping used to evaluate physical locality

Figure 4.5 shows a static mapping in which threads are allocated in an H-tree manner across the chip. The intention here is to map threads with consecutive IDs to cores in close proximity of each other.

Figures 4.2c and 4.3c present the results from the proximity hit evaluation for this H-tree mapping. The static H-tree scheme reclaims much of the locality that the random mapping failed to capture. Importantly, even with a static non-ideal mapping and added restrictions of the 2D topology, it is possible to achieve a high number of proximity hits for many benchmarks.

Table 4.1 summarises the performance of the realistic mappings when compared to the ideal mapping. The H-tree mapping outperforms the random mapping in all but one case. For some benchmarks the margin is over 40%. However there is a noticeable difference in performance between the SPLASH-2 and Parsec benchmarks. The results show that for Parsec benchmarks, it is very important to have an ideal mapping in order to extract good locality. Neither the random or H-tree mappings do a good job of capturing the available physical locality, and this suggests that further work into more sophisticating mappings may improve results seen for Parsec benchmarks.

Program	Random	H-Tree	Difference
barnes	81%	95%	13%
cholesky	56%	79%	23%
fft	86%	97%	11%
fmm	57%	87%	31%
lu	57%	76%	19%
ocean	35%	84%	49%
radiosity	90%	99%	9%
radix	30%	91%	61%
raytrace	62%	82%	21%
volrend	82%	92%	11%
waternsquared	80%	96%	15%
waterspatial	83%	93%	11%
blackscholes	96%	100%	5%
bodytrack	85%	87%	2%
canneal	94%	97%	3%
dedup	64%	68%	5%
facesim	64%	67%	3%
ferret	44%	45%	1%
fluidanimate	26%	32%	5%
freqmine	87%	83%	-4%
streamcluster	54%	96%	41%
swaptions	74%	77%	3%
vips	69%	71%	2%
x264	82%	85%	3%

Table 4.1: The hit rates of Random and H-Tree mappings normalised to that of the Ideal mapping. Results are shown for a snoop width of 4

CHAPTER 5

Proximity Coherence

This chapter introduces two of the contributions made by this research. First, the Proximity Coherence protocol, a scheme in which L1 load misses are optimistically forwarded to nearby caches rather than always being indirected via a directory structure, and second, the dedicated wiring links over which the protocol runs.

5.1 Introduction

To effectively utilise the increasing number of transistors available in modern fabrication technologies, the semiconductor industry is moving to many-core architectures [1; 70; 75]. These architectures provide better scalability than monolithic single core superscalar architectures. While a many-core processor behaves much like a multi-node system implemented on a single chip, important differences exist: the amount of storage available on-chip is much more restricted, and the communication latencies are considerably lower. Furthermore, the close proximity of processing and storage elements allows for optimisations that were previously unattractive in a multi-node system. Many-core processors are unconstrained by the packaging and interconnect latencies of larger multi-node machines, suggesting many possible architectural advances.

This chapter investigates Proximity Coherence, a protocol in which the private caches of neighbouring cores are probed upon a cache miss. Instead of immediately sending a message to the directory, a core first asks neighbouring caches for a copy of the re-

quired line. The core sends a request to the directory only if all neighbouring caches reply that they do not have a copy of the data. Implementing this scheme in multi-node system would be impractical, as the latencies to snoop another cache would be of the same magnitude as going immediately to the directory. Moreover, in the case that no neighbouring cache can provide the data, the request must still be sent to the directory, drastically increasing the service time.

However, in a many-core system, the communication costs are different. Messages can be carried between neighbouring cores using dedicated point-to-point links, minimising both latency and energy costs. The overhead of probing a neighbouring cache then becomes only a few cycles. This delay is insignificant compared to the service time of a request that is routed to a directory.

This work presents a novel extension to a standard MESI cache protocol [65] that implements the snooping mechanism described and provides lower cache miss latencies. The concept of a proximity cache hit is introduced, where data is provided by a neighbouring cache without involving the directory. Additionally, this work proposes the use of lightweight graph structures embedded into the private cache lines to maintain coherence despite the lack of global knowledge at the directory. All proximity coherence messages are carried to neighbouring cores on new, dedicated, point-to-point links – an implementation made possible by the close proximity of processing elements and the abundance of wires available in many-core architectures.

5.2 Motivation

5.2.1 Proximity Hits

When a memory access misses in the cache of a traditional chip-multiprocessor, the request is forwarded to a directory structure. In some cases, the data is already present in a different private cache in the system. The baseline MESI protocol deals with this scenario in one of two ways, depending on whether a private cache has exclusive ownership (states **E** or **M**) of the line. In the first case, providing that no private cache has exclusive ownership for that line, the data is returned from the L2 to the original requester. In the second case, the directory sends a request to the exclusive cache, instructing it to send the data to the requesting cache. In both situations, it is possible to bypass the indirection to the directory and ask private caches already containing the line to provide the data immediately.

This work proposes a scheme in which cache lines are requested directly from other private caches without contacting the directory, avoiding the aforementioned indirection. This process is referred to as *snooping* another private cache. A situation where a processor misses in its local private cache but receives at least one copy of the requested data directly from another private cache is declared a *proximity hit*.

5.2.2 Baseline Architecture

Proximity Coherence exploits the principle that data may be available in other private caches in the system upon a miss in a processor's local private cache. Chapter 4 presents an evaluation of this new physical locality, concluding that a considerable number of cache misses can be satisfied by snooping only four other private caches in the system.

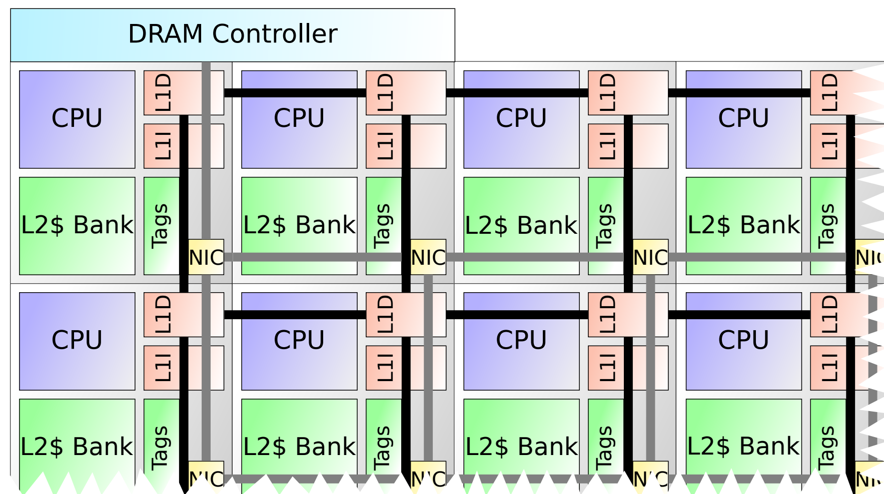


Figure 5.1: Top left corner of a tiled many-core processor. Grey connections show the global on-chip interconnect. Black connections show the proximity links between the L1D caches.

However, to design an effective protocol enhancement it is necessary to look beyond the idealised configuration used earlier.

This following work evaluates a more realistic architecture than the idealised machine used in Chapter 4. Figure 5.1 shows a corner of the processor, composed of 32 processing tiles arranged in an 8x4 grid. Each tile consists of a processing core, a private L1 cache, a single bank of the interleaved, shared L2 cache and a network interface that connects the tile to the global on-chip network. Four memory controllers are placed in the corners of the chip. The L2 cache contains a directory that uses a MESI protocol to maintain coherence across all private L1 caches in the system.

Due to constraints on wiring resources and limited cache ports, any implementation of Proximity Coherence must select a sub-set of processors in which to snoop for data. This work refers to the size of this subset as the *snoop width*. The tiled nature of the architecture means the results when snooping only 4 neighbouring caches are of most interest. Figures 4.2 and 4.3 show that snooping just four caches captures the large

majority of all possible hits. This suggests that the parallel benchmarks examined can have stable sets in which data is shared, allowing for good proximity hit performance through the use of correct thread mappings and network topologies.

5.2.3 Concurrent Proximity Requests

The forwarding of cache misses to adjacent processors increases the strain placed on the read ports of private caches. Although the probability of generating a proximity message requiring read port access is low, a single cache could be expected to serve up to the four concurrent requests from adjacent tiles. Trace analysis shows that the likelihood of this happening is extremely low – averaged across all benchmarks used here, 99.39% of proximity requests encountered no contention from other proximity requests. 0.6% of requests encountered contention from a single concurrent request, with three-way and four-way contention making up the final 0.01%. Such a low probability of contention permits the reuse of existing cache read ports and a simple arbitration mechanism, with no fear of degrading performance through the stalling of proximity messages.

5.2.4 Energy Considerations

Research in the network-on-chip field [75] has shown that the energy cost of network routers will inevitably comprise a significant portion of total system demand. As a consequence, schemes that reduce network hop traversals are becoming increasingly attractive. Additionally, advanced on-chip router energy consumption is now comparable to an L1 cache access [7; 50; 74]. Importantly, this can offset the use of the additional L1 cache accesses generated by Proximity Coherence through reduce network utilisation.

5.2.5 Summary

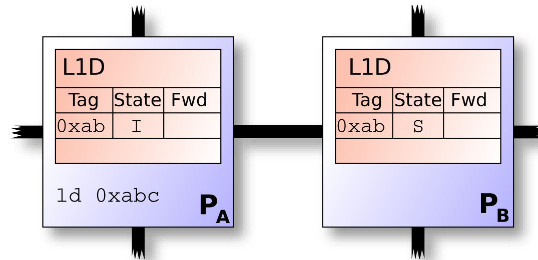
Rising communication costs and the demand for high performance data sharing motivates the extension of existing cache coherence protocols to exploit the physical locality of shared data.

5.3 Proximity Coherence

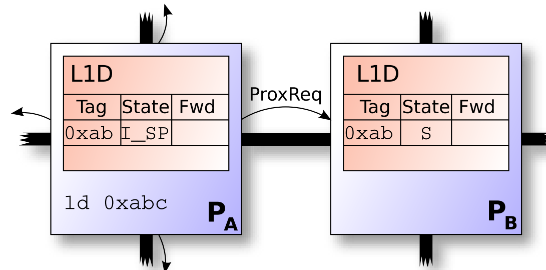
Proximity Coherence is built on the concept that a core snoops its four neighbouring caches before sending a request to the directory. This work refers to this as a *proximity-request*. If a snooped cache can provide the data, it performs a cache-to-cache transfer to the requester and marks the data as forwarded. If any neighbouring caches supply the requested data, then the original cache miss is classified as a *proximity-hit*. These cache-to-cache transfers use point-to-point links between neighbouring cores, rather than the packet switched, global on-chip network. Due to the critical nature of proximity requests from adjacent nodes (the adjacent processor may be stalled on the outstanding cache miss), they are prioritised when arbitrating for cache read ports.

Forwarding data in this way presents design challenges, as the directory is not aware of the additional sharers. In order to maintain coherence, modifying the cache coherence protocol is necessary to provide the following mechanisms:

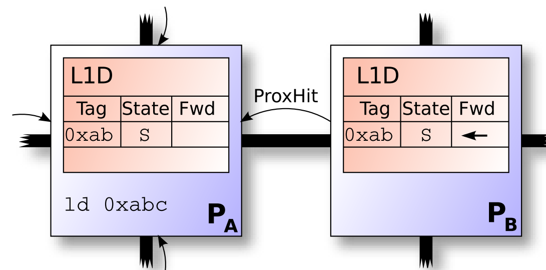
- When an L1 cache replaces a cache line that has been forwarded, it sends an L1_UPDATE_S (Update Sharer) message to the directory. The message contains a list of the cores to which the replacing cache has forwarded the data. To avoid incoherent data being held in the system, it is required that the directory to acknowledge this message. A similar mechanism is already used in the baseline MESI protocol when an L1 cache evicts a dirty cache line.



(a) P_A performs a load operation, which misses in its L1 cache. P_B has a copy of this data in its local cache with read permissions.



(b) Instead of contacting the directory, P_A sends PROXREQ messages to neighbouring cores. These messages are sent using direct point-to-point links.



(c) P_B can supply the data to P_A and replies with a PROXHIT message. In addition, it records in the 4-bit fwd-vector that it has forwarded the data. P_A obtains the data through a *proximity hit*.

Figure 5.2: Example of a *proximity hit*.

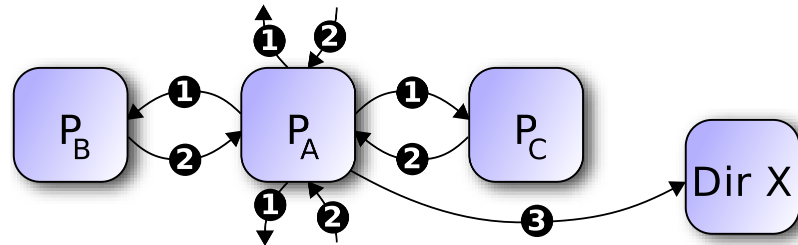


Figure 5.3: Example of a *proximity miss*. P_A misses in its local cache and sends out four PROXREQ messages to its neighbouring cores (step ❶). Since none of these tiles can provide the data, they all respond with a PROXMISS message each. This situation is called a *proximity miss* (step ❷). P_A now sends a GETS message to the directory in order to request the data (step ❸).

Due to silent evictions of shared data, it is possible that the L1_UPDATE_S message will contain cores that no longer hold a copy of the data. This is not an issue, as the MESI baseline protocol dictates that invalidates received for non-present data are immediately acknowledged.

- When an L1 cache receives an INVALIDATE message it is necessary to propagate this message to any cores to which it has forwarded the cache line. After the cache has received all acknowledgements, it can then acknowledge the original INVALIDATE message. As the propagated messages (PROXINV) can only be sent to neighbouring cores, they are sent using the same direct links as proximity-requests.
- If a core requires exclusive access to a cache line that it has already forwarded, all forwarded copies must be invalidated and an UPGRADE message sent to the directory. These events can be performed in parallel, speeding up the invalidation process.

5.3.1 An Example of Proximity Coherence

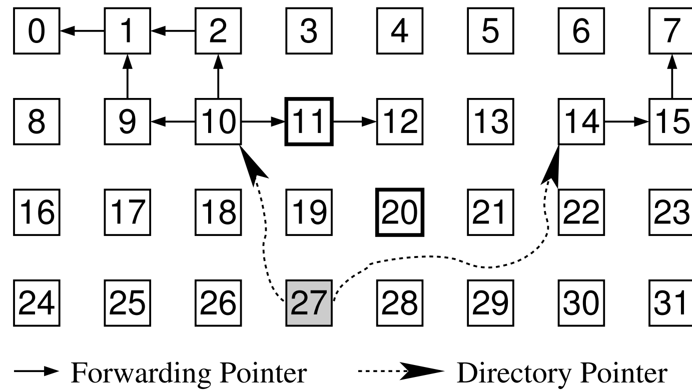
Figure 5.2 shows the detailed behaviour of the Proximity Coherence protocol when a load operation misses in an L1 cache. P_A issues a load to address `0xabc`, but the

corresponding line `0xab` is not valid in its cache. P_B has a valid copy of this line in state Shared (Figure 5.2a). Instead of sending a request to the directory, P_A sends out 4 *Prox Requests* to its neighbouring cores and moves the line into a transient state (Figure 5.2b), which indicates that the cache is awaiting replies from all proximity requests. Since P_B has a valid copy of line `0xab`, it replies by sending a PROXHIT message containing the data and marks the cache line as forwarded to its left neighbouring core (Figure 5.2c). The forwarding is recorded in a 4-bit entry, encoding the forwarding state for each of the four neighbouring cores. The requesting core will write the data that arrives first to its private cache. As there is no acknowledgement of a proximity hit from the requester, every core that provided the data will mark its cache line as forwarded. Hence, for a single address, several cores can point to a single requester.

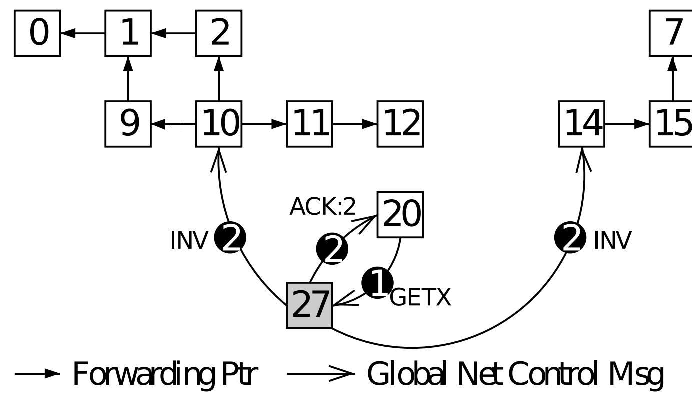
Figure 5.3 shows the actions taken if a load operation does not hit in any of the neighbouring caches. As before, P_A sends out proximity requests to its neighbouring caches (step ❶). As none of the caches contains a copy of the data that can be forwarded to P_A, they all respond with a PROXMISS message (step ❷). After P_A has collected all the replies, it sends a GETS message to the directory responsible for this cache block (step ❸).

5.3.2 Invalidations

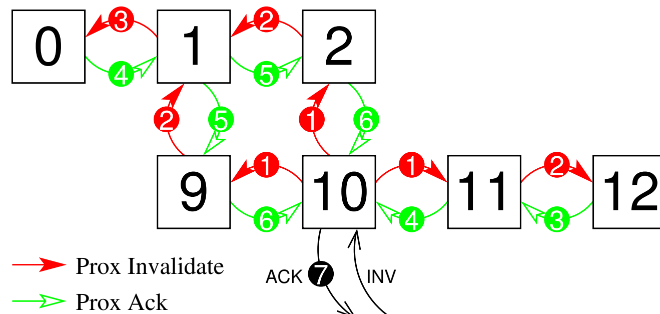
As Figure 5.2 shows, any cache that forwards data to another core records this action in the forwarded vector for that line. This process can occur several times, forming an acyclic *forwarding graph*, as depicted in Figure 5.4a. As a cache must hold a line to be the source of a forwarding pointer, it is impossible to form a cycle in the graph. Cores P10 and P14 originally received their data from the directory (located for this particular address in core P27). Core P10 has forwarded the data to cores P2, P9 and P11, while



(a) Initial situation: the directory is aware of 2 sharers, P10 and P14. P10 is the root of the left half of the forwarding graph, while P14 is the root of the right half. Tile P20 and P11 are emphasised since they will start the process that will lead to the invalidation of all sharers.



(b) P20 sends a GETX message to the directory in order to gain exclusive ownership of the cache line. The directory responds by sending INV messages to the known sharers and an ACK:2 message to P20. All these messages are sent over the global on-chip network.



(c) Invalidation of the left part of the forwarding graph. The PROXINV and PROXACK messages are sent over the direct links between neighbouring cores.

Figure 5.4: Example of external invalidations. For this cache line, the directory is located in core P27, indicated by solid grey shading.

core P14 has forwarded it to core P15. These cores in turn have forwarded the data to other cores, as indicated by the forwarded arrows. When core P1 has requested the data, both cores P2 and P9 return a copy. Therefore, both cores hold a record that they forwarded the data to core P1. As the directory has sent the data to only cores P10 and P14, it holds pointers to only these cores. For this reason, on an invalidation, it is necessary to follow the forwarded links in order to reach and invalidate all copies of the data. The following paragraphs present examples of the two types of invalidations found in Proximity Coherence:

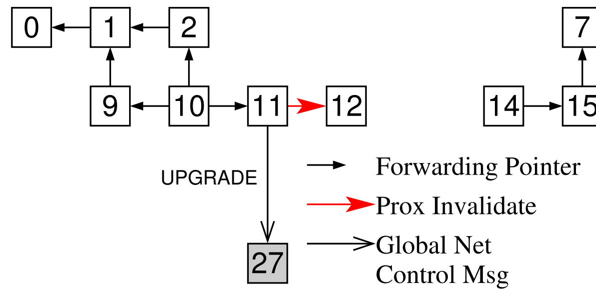
External Invalidations occur when a core, which is not part of the forwarding graph, needs to modify shared data. In Figure 5.4b, core P20 requires exclusive access to a cache line. As in a normal MESI directory protocol, core P20 sends a GETX message to the directory (step ❶). The directory responds by sending invalidates to the two sharers it has knowledge of (cores P10 and P14) and in parallel notifies core P20 that it should wait for two acknowledgements (step ❷). The protocol now diverges from the standard MESI behaviour. Before cores P10 and P14 can reply with an acknowledgement, they have to invalidate the cores to which they have forwarded the data. Figure 5.4c shows how core P10 invalidates these cores (core P14 acts in a similar way, but for simplicity only core P10 is shown). Core P10 sends PROXINV messages to cores P2, P9 and P11 (step ❶). Since these cores also forwarded the data, they too must send PROXINV messages (step ❷). A special case is core P1, since it received data from both core P2 and P9. As such, P1 will potentially receive two PROXINV messages before it receives confirmation from core P0, to which it forwarded the data. To remember the cores to which the PROXINV messages were sent, the function of the forwarding vector is changed; instead of keeping track of to whom the cache line has been forwarded it keeps now track which cores send a PROXINV message. When the end of the forwarding chain

is reached, the final core replies with a PROXACK message (see core P0 in step ❹). This in turn causes the previous core in the chain to generate a PROXACK message. Once all PROXACK messages have been collected by core P10, it sends an ACK message over the global on-chip network to the new exclusive owner of the cache line (step ❼). The remaining actions are identical to those in a standard MESI protocol.

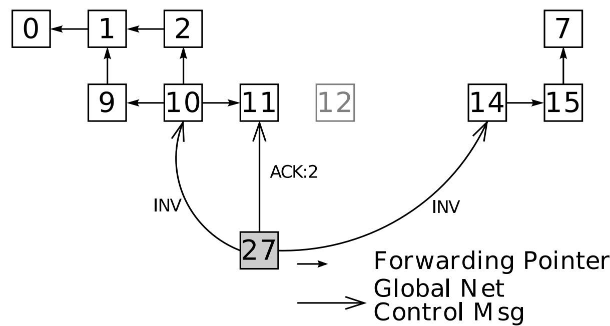
Internal Invalidations occur when a core, which is part of the sharer graph, needs to modify shared data, such as core P11 in Figure 5.5a. Core P11 sends an UPGRADE message to the directory to request exclusive access to the cache line. Since P11 also has forwarded the cache line to other cores, it sends PROXINV messages to these cores (see Figure 5.5a). For simplicity, the diagram shows a situation in which the PROXINV messages are acknowledged before the GETX message is processed by the directory. However, this is not a requirement of the protocol; the events are allowed to occur in any order. The directory responds in the standard manner sending out two INV messages and one ACK message that tells core P11 how many sharers there were (see Figure 5.5b). Once core P10 and P14 have received the invalidate messages, they send out PROXINV messages to cores P2, P9, P11 and P15. As such, P11 will receive an invalidate message, even though it originated the request. To prevent P11 from invalidating itself, the PROXINV messages must contain a field identifying the original requester. Therefore, if a core receives a PROXINV message for which it is the originator, it can ignore the message and reply with a PROXACK.

5.3.3 L1 Cache Replacements

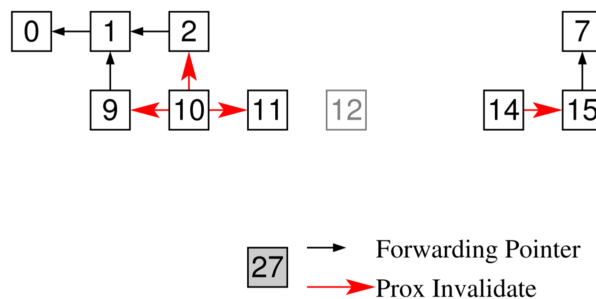
At any time during the life of a forwarding graph, a participating cache can evict its data. If the protocol were to behave as a standard MESI protocol, and perform a silent



(a) P11, which is part of the sharer graph, sends an UPGRADE message to the directory in order to gain exclusive ownership of the cache line. Since it also has forwarded the data to its right neighbouring core, it also sends a PROXINV message to this core – proactively invalidating it.

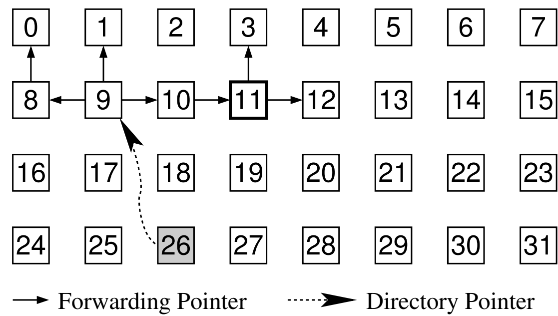


(b) By the time the directory processes the UPGRADE request, core P12 has already invalidated its copy of the data and is shown in grey. The directory responds by sending INV messages to the known sharer and ACK:2 message to P11.

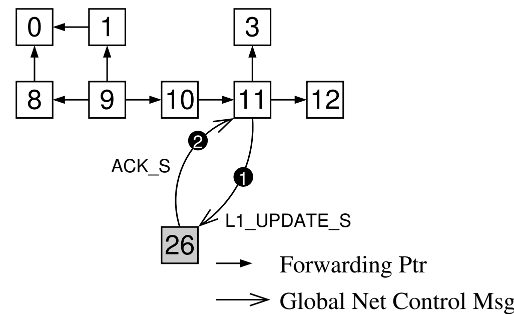


(c) After core P10 and P14 have received the INV messages, they send PROXINV messages to the cores they have forwarded the data to. Since P11 is the originator of the request, it ignores the PROXINV by acknowledging it without invalidating its copy of the data.

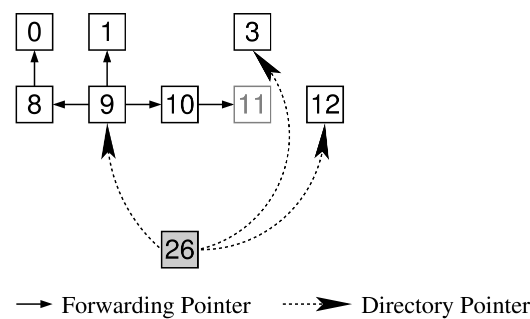
Figure 5.5: Example of internal invalidations. For this cache line, the directory is located in core P27, indicated by solid grey shading.



(a) Initial situation: the directory is aware of 1 sharer, P9. P9 is the root of the forwarding graph. Core P11 has forwarded the cache line to core P13 and P12. It now wants to replace the cache line.



(b) In order not to break the forwarding graph, it sends an L1_UPDATE_S message to the directory. The directory adds the sharers contained in this message to its sharer vector and acknowledges the receipt with an ACK_S message.



(c) Final situation: core P11 has invalidated its copy of the cache line, shown in grey. The directory is now aware that core P3 and P12 have a copy of the data and holds a direct pointer to them.

Figure 5.6: Example of an L1 replacement in case of forwarded data. For this cache line, the directory is located in core P26, indicated by solid gray shading.

eviction, the graph would be irreparably broken. To prevent this, Proximity Coherence modifies the mechanics of L1 replacements:

- If the cache has not forwarded the data to any other core, it behaves as in the standard MESI protocol and simply replaces the cache line without informing the directory. If it later receives a PROXINV message for the replaced address from any neighbouring cores or an INV message from the directory, it acknowledges the message.

- If the core has forwarded the data, then it must inform the directory of the other sharers before it can replace the cache line. This action is similar to an L1 cache trying to replace a cache line that contains dirty data: before the cache line can be replaced, it has to be written back to the L2 and the directory has to be informed. Proximity Coherence uses the same simple mechanism. This mechanism also deals with cases when, during an L1 replacement, another L1 tries to gain exclusive access to the data and wins the arbitration at the directory.

Figure 5.6 illustrates such a scenario. The starting situation is shown in Figure 5.6a. The directory in core P26 is only aware that core P9 has a copy of the data, while core P11 wants to perform a replacement, having forwarded the data to cores P3 and P12. P11 sends an UPDATE_S (Update Sharer) message to the directory (step ❶). Upon receiving this message, the directory adds the sharers contained to its sharer vector and sends an ACK_S message back to core P11 (step ❷). To prevent protocol races against external invalidations, P11 must retain the sharer information for the cache line until the ACK_S message is received. Figure 5.6c shows the situation after the replacement: the directory is now aware that cores P3, P9 and P12 have a copy of that cache line. Core P10 maintains a forwarded pointer set towards core P11 but this has minimal impact;

P11 might receive a spurious PROXINV message for the replaced address, but can simply ignore this.

5.3.4 Forwarding from Modified and Exclusive

In addition to supporting *Load on S* forwarding described so far, Proximity Coherence also allows data to be forwarded from a line that is held with write permissions. Forwarding is supported from the **M**odified and **E**xclusive states of the baseline MESI protocol.

When a proximity-request is received for a cache line held in the **M** or **E** states, the data is returned as a proximity-hit and the cache line is moved immediately to a new **F**orwarded state. This **F** state indicates to the forwarding cache that the line's permissions have been downgraded, without the directory's knowledge, to read-only access. A processor holding a line in the **F** state is responsible for any copies it forwarded on. Should the core receive an invalidate, it must invalidate all copies of the data forwarded to adjacent processors.

Supporting forwarding in this way is important, as when a line is first loaded into the system it arrives with exclusive permissions in the requesting private cache. Hence, without the addition of *Load on M* forwarding, the first proximity-request is guaranteed to miss, creating unnecessary traffic to the directory.

In a situation similar to that described in Section 5.3.3, the forwarding graph can be broken into two parts, requiring the replacing cache to send an L1_UPDATE_S message to the directory. For this reason, maintaining the read-sharers vector in the directory state machine is essential, even when the line is believed to be held with exclusive access in a private L1 cache. No extra storage is required to support this extension. In the special case that the replacing cache is the root of the forwarding graph, a message is

returned to the directory containing both the forwarding vector and, if the line is dirty, the data. Again, the directory state machine is augmented to allow for such messages to be processed.

Before a processor can write to a cache line that is held in the **F** state, the processor must reacquire exclusive access. This is achieved by invalidating the forwarded read-access copies of the data using proximity invalidates, and in parallel sending an UPGRADE request to the directory. When all forwarded copies are invalidated and confirmation is received from the directory, the cache line returns to **M**odified and the write completes.

5.3.5 State Machine Description

This section lists the states for both the L1 and L2 structures in Proximity Coherence. Not all of the additional states are used to resolve protocol races — many are added to track progress through a state transition, for example waiting for four proximity-acknowledgments and a single directory-acknowledgment. In an optimal implementation, these states could be collapsed to minimise the number of bits required to enumerate the current state of each cache line.

5.3.5.1 L1 States

The stable L1 states used in Proximity Coherence are very similar to the original MESI protocol. The only addition is the new F state, which is used to denote that a node used to hold write permissions, but has since forwarded data over proximity links. Figure 5.7 shows how the F state interacts with the original protocol, and Table 5.1 enumerates all of the stable states.

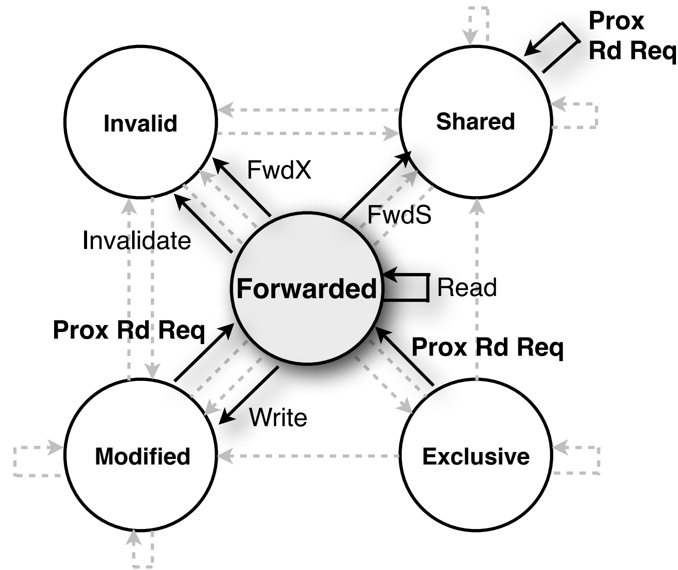


Figure 5.7: A diagram of the stable L1 states in the Proximity Coherence. Standard MESI transitions are greyed out for clarity.

NP	Not present in either cache
I	Invalid
S	Shared
E	Exclusive
M	Modified
F	Forwarded. Data was previously held in state M/E, but received prox-req and relinquished write permissions to enable forwarding to requester

Table 5.1: The stable states implemented in the GEMS model of each L1 cache.

The complexity of Proximity Coherence is found primarily in the transient states of the L1 state machine. Table 5.2 lists the temporary states used to move between the stable states listed above. For clarity and debug purposes, this is an unoptimised state machine.

IS	Issued GETS, have not seen response
IM	Issued GETX, have not seen response
SM	Issued GETX upgrade request, have not seen response
IS_I	Issued GETS but received invalidate before response
M_I	Replacing dirty data, waiting for acknowledgement from directory
E_I	Replacing dirty data, waiting for acknowledgement from directory
IS_P	Sent prox-requests, waiting for responses
IS_PI	Sent prox-requests, received invalidate from directory while waiting for responses
IS_PH	Sent prox-requests, received one more more prox-hits, waiting for other responses
S_PM	Sent prox-invalidates to forwarded, waiting for acknowledgements
S_PIP	Received prox-invalidate, sent out prox-invalidates, waiting for acknowledgements
S_PRI	Replacing forwarded line, sent update-sharers message to directory, waiting for dir-ack
S_PIP_R	Sent update-sharers message to directory, received prox-invalidate before dir-ack
SM_PID	Received diretory-invalidate-forward request while waiting for upgrade to M
SM_PIP	Received prox-invalidate-forward request while waiting for upgrade to M
S_PUI	Sent update-sharers to directory, but received directory-invalidate
F_R	Replacement of line in state F, waiting for write-back ack from directory
F_M	Line in F needs to return to M. Waiting for prox-invalidate-acks and dir-ack
F_I	Received directory-invalidate while in state F. Sent prox-invalidate, waiting for all prox-acks
F_FI	Received request to forward exclusive permissions on, invalidating all forwarded copies
F_R_I	Replacing line, received directory-invalidate before acknowledgement
F_M_FWDS	Sent upgrade request, but received forward-S-request from directory, lost race so invalidate forward copies and forward data according to forward-S-request
F_M_FWDX	Sent upgrade request, but received foward-X-request from directory, lost race so invalidate forward copies and forward data according to forward-X-request
F_M_PAA	Sent upgrade request, finished invalidating forwarded copies, waiting for directory-ack
F_M_I	Sent upgrade request, but received directory invalidate. Lost race, so invalidate
F_M_PAA	Sent upgrade request, finished invalidating forwarded copies, waiting for directory-ack
F_R_FWDS	Replacing line, received forward-S-request before directory-ack
F_R_FWDX	Replacing line, received forward-X-request before directory-ack
F_R_FWDX_PAA	Replacing line. Received forward-S-request before directory-ack. Collected all prox-acks.
F_R_I_PAA	Replacing line, received invalidate from directory before ack. Collected all prox-acks
F_R_FWDX_DIR	Replacing line, received forward-x-request, received directory-ack of replacement
F_R_I_DIR	Replacing line, received invalidate, received directory-ack of replacement
F_M_D_PEND	Requested upgrade. Waiting for directory-ack
F_M_D_COMP	Requested upgrade. Received for directory-ack
S_PID_R_PAA	Replacing line, but received directory invalidate. Received all prox-acks
S_PIP_R_PAA	Replacing line, but received prox invalidate. Received all prox-acks

Table 5.2: The transient states implemented in the GEMS model of each L1 cache.

5.3.5.2 L2/Directory States

The L2 state machine does not require any additional states to support Proximity Coherence, hence Tables 5.3 and 5.4 contains only those states already present in the original MESI protocol. Additional work is added to state transitions as described in Section 5.3.4

State	Description
NP	Not present in any cache
SS	Shared. Present in one or more L1 caches
M	Modified in L2, but not present in any L1 cache
MT	Modified in a local L1, assume L2 copy is stale

Table 5.3: The stable states implemented in the GEMS model of each L2 cache.

State	Description
M_I	L2 cache replacing. Received all L1-acks, sent dirty data to memory, awaiting ack
MT_I	L2 cache replacing. Waiting on data from exclusive owner L1
I_I	L2 cache replacing clean data. Wait for L1-acks and silently evict line
S_I	L2 replacing dirty data. Waiting for L1-acks before writing back to memory
SS_MB	Blocked for GETX request, previously in state SS
MT_MB	Blocked for GETX request, previously in state MT
M_MB	Blocked for GETX request, previously in state M
MT_IIB	Blocked for GETS request, previously in state MT
MT_IB	Blocked for GETS request, previously in state MT, waiting for data
MT_SB	Blocked for GETS request, previously in state MT, waiting for unblock

Table 5.4: The transient states implemented in the GEMS model of each L2 cache.

5.3.6 Race Conditions

The protocol races encountered in Proximity Coherence are similar in type to those found in the original MESI protocol. The additional complexity is introduced by the fact that there are both prox-link, and global network versions of most messages. This greatly increases the number of potential races, but as already discussed, suitable changes can be made to the L1 state machine to deal with all possible scenarios.

Proximity Coherence was tested for robustness against potential race conditions us-

ing the GEMS SLICC protocol tester. This tool uses artificial traffic patterns specifically designed to exposed protocol races, and reports any errors it encounters. Ideally it would be possible to formally verify the functionality of the protocol using a tool such as Murphi [29]. However this is a challenging task, particularly when hoping to draw conclusions about the behaviour of a system with a realistic number of processors. Such an analysis is left for future work.

There are seven basic types of race:

1. GETS/GETX against INVALIDATE
2. UPGRADE against INVALIDATE
3. Write-back of a line in **M** state against FWDS, FWDX or INVALIDATE
4. PROXREQ against INVALIDATE
5. UPDATE_S against INVALIDATE
6. Write-back of a line in **F** state against FWDS, FWDX or INVALIDATE
7. UPGRADE for a line in **F** state against FWDS, FWDX or INVALIDATE

Races 1, 2 and 3 are already present in the standard MESI protocol and are resolved in exactly the same way.

Race 1 happens if a core previously cached a line as read-only, silently replaced it and now wants to cache it again. As the directory is the not aware of the silent replacement, it will send an INVALIDATE message if another core requests exclusive access at the same time. The core will acknowledge the INVALIDATE, and the GETS/GETX will be queued at the directory until it can be processed. Race 2 happens if a core wants to upgrade a line to exclusive access, while at the same time another core is trying to do the same. The core that receives an INVALIDATE knows it has lost the race, and must clear the

data from the cache. Similar to race 1, the original UPGRADE message will be queued at the directory and serviced when possible. Race 3 occurs if a core has exclusive access to a line and needs to replace it, while at the same time either another core is requesting access to this line or the directory has to replace the entry due to a conflict miss. The core must acknowledge the invalidation, and wait for its original request to be queued and serviced by the directory.

The remaining races, while unique to Proximity Coherence, are very similar to races found in a standard MESI protocol, and as such can be resolved using similar mechanisms.

5.3.6.1 PROXREQ against INVALIDATE

This situation arises when a core silently replaces a L1 cache line and later tries to read from same address again. When issuing the read, the core sends a *ProxReq* to its neighbours. However before receiving a response to the proximity request, it receives an INVALIDATE or PROXINVmessage. This situation is identical to race 1, and can be resolved in the same way as the standard MESI protocol resolves this race.

5.3.6.2 UPDATE_S against INVALIDATE

This race occurs when core X has forwarded a cache line from state **S**, moving the line to state **S**, and now wants to replace the data. Core X informs the directory of the cores it forwarded the data to (see Section 5.3.3). Another core Y then requests exclusive access to the same cache line. The directory sends INVALIDATE messages to all known sharers and number of acknowledgements to the requestor.

Proximity Coherence deals with this situation in the following way: the directory will receive the UPDATE_S message after it has informed the requestor of how many acknowledgements to expect. It is not possible for the directory to send additional

INVALIDATE and reliably update the number of acknowledgements that core Y should expect. Such a message would race against ACK messages resulting from both sets of INVALIDATE messages. Thus, core Y could assume that the transaction is complete, while it still has to wait for additional ACK messages.

This means that core X must resolve the race when it receives the INVALIDATE message. In order to do so, it still has to remember to which cores the cache line was forwarded. It invalidates these cores via PROXINV messages. After all PROXACK messages have been collected, it can acknowledge the INVALIDATE message and discarding the cache line completely. This way the number of ACK messages that core Y receives matches the number reported by the directory.

Finally, the directory cannot respond with an UPDATE_S_ACK message. Otherwise, this message would race against the earlier sent INVALIDATE message. If it arrives before the INVALIDATE message, then core X would discard the required sharer information. Instead, the directory replies with an UPDATE_S_NACK message.

5.3.6.3 Write-backs from State F

A core X that replaces a line in state **F** must inform the directory of the cores to which it has forwarded the data (see Section 5.3.4). Until it receives confirmation from the directory, the core has to keep a copy of the cache line and the forwarding vector. In this situation, it is possible that another core Y sends a GETS, GETX or UPGRADE request to the directory as well. The type of message determines how this races is resolved (if the message from core Y arrives first):

- **GETS:** This situation is very similar to race 3. The directory responds by sending a FWDS message to core X and enters a blocking state. Upon receiving this message, core X will treat it as an implicit acknowledgement of its write-back message and

forward a copy of the replaced cache line as specified in the FWDS. Once core Y receives the data, it will send an unblock message to the directory. This behaviour is identical to race 3. The directory will receive the write-back message from core X at some point. In addition to the notification that the data has been written back (identical to the behaviour in a standard MESI protocol), it will update its sharer list.

- **GETX:** The directory responds by sending a FWDX message to core X and enters a blocking state. In addition, the directory will send INVALIDATE messages to other cores that might have been added for this address by previously received UPDATE_S messages. The directory will ignore the write-back message, as core Y requested exclusive access. Once core X receives the FWDX message, it will send out PROXINV messages to cores it forwarded the data to. After collecting all PROXACK messages, it will forward the data to core Y and invalidate its copy.
- **UPGRADE:** This situation is similar to the GETX case. However, instead of forwarding the data to core Y, core X will just invalidate it after collecting all PROXACK messages. The data does not have to be forwarded, as core Y already has a valid copy.

5.3.6.4 UPGRADE of a line in F

This situation arises if a core X wants to regain exclusive access for a line in state **F**. It sends out an UPGRADE message to the directory to invalidate sharers that can no longer be reached by the proximity links (see Section 5.3.4). Another core requests access or exclusive access to the same cache line. This situation is a combination of race 2 and race 5. Unlike race 5, core X has already sent PROVINV to its neighbours. After it has

collected all ProxAck messages, the situation is identical to race 2 and will be handled as such.

5.3.7 Hardware Costs

Implementing Proximity Coherence incurs only a small hardware overhead. In contrast to similar works [31; 45], no additional complexity is required in either the processor or network routers. First, the protocol needs additional wires for the point-to-point links that are used for proximity requests. These wires are, on average, the length of one tile and do not require deep buffering. Flow control is provided by a simple not-ready wire applying back-pressure. Moreover, there are a large number of such wires available in modern fabrication processes [28; 42], particularly in wiring channels between tiles. Second, each cache line needs additional bits to store where the cache line has been forwarded. In this particular implementation of the scheme, data can be forwarded to any of the four neighbouring cores requiring an additional four bits per cache line. This increases the stored information in each L1 cache by less than 1%, assuming 64 byte cache lines with 51 bit tags. Finally, as established in Section 5.2.3, it is not necessary to increase the number of cache read ports. All new structures used by Proximity Coherence are distributed and will scale well to larger core counts without incurring additional hardware overheads.

CHAPTER 6

System Evaluation

6.1 Introduction

When investigating changes to cache coherence protocols it is vital to evaluate the performance impact using execution driven simulators. The behaviour of parallel programs is often highly unpredictable when implementing large architectural changes, so to ensure that the expected benefits are delivered, this work uses the GEMS toolset [58] to run as many benchmarks as possible on a chip-multiprocessor model using the new Proximity Coherence protocol.

The results show that the new protocol is able to reduce the latency of load misses by up to 33%, and 17% on average, resulting in overall execution time improvements of up to 13%, for the chosen subset of benchmarks. In addition to providing these performance benefits, Proximity Coherence also reduces network-on-chip traffic by 19% and cache hierarchy energy consumption by up to 30%.

6.2 Evaluation Setup

To evaluate the performance benefits of the Proximity Coherence scheme, a cycle accurate version of the protocol was implemented. This section of work uses a different ISA and OS to the work in Chapter 3. This requirement was enforced by the execution driven infrastructure used to evaluate the design. Unfortunately at the time the research

Processors	32 Sparc V9 cores, 3 GHz, single-issue, in-order, non-memory IPC = 1
OS	Solaris 9
L1 cache	32 kB per core, split I/D, 4 way associative, 2 cycles latency, 64 byte lines
L2 cache	8 MB, 32 banks interleaved, 8 way associative, 16 cycles latency, 64 byte lines
Memory	1GB, 4 banks, 250 cycles latency
Directory	L1 tag replication, 32 banks interleaved, MESI protocol
Network	8x4 mesh topology, 2-cycle routers, 1-cycle link latency, 36 bytes wide
Prox-Links	1-cycle link latency, 36 bytes wide, single-depth buffers

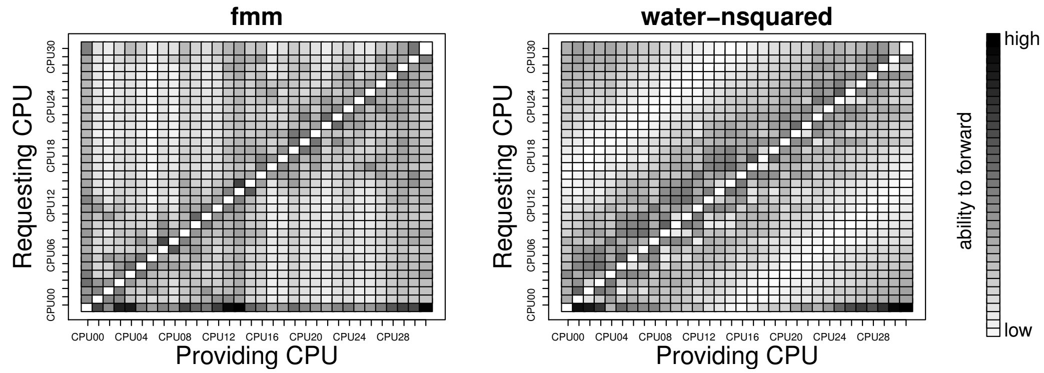
Table 6.1: Parameters used in the full-system simulation to evaluate Proximity Coherence.

was conducted, there were no suitable academic simulators that supported the x86 ISA used in earlier work.

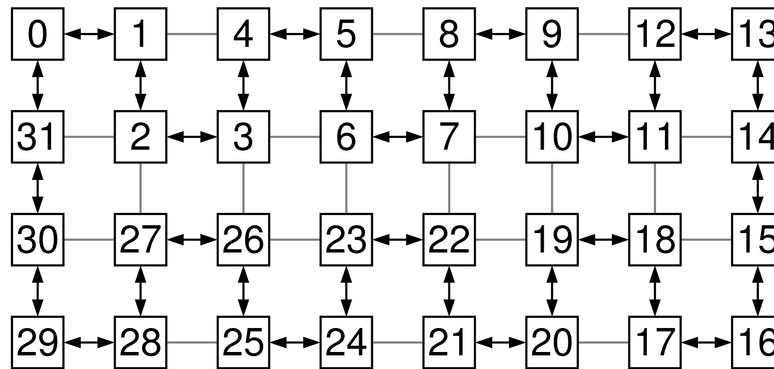
6.2.1 Simulation Parameters

For full-system simulation, this evaluation uses Virtutech Simics [57] and the Wisconsin GEMS tool set [58]. These tools provide full OS support and a customisable memory model. The GEMS SLICC language is used to define the extended state machine with all transient states and the necessary storage additions to hold forwarding vectors for each cache line. The protocol has been thoroughly stress-tested using the supplied SLICC protocol tester to check for race conditions and consistency violations. As described in Section 5.3.7, Proximity Coherence uses an augmented version of the existing GEMS network model with fast point-to-point links between neighbouring tiles.

Table 6.1 lists the parameters of the simulated system. These parameters are in line with recently proposed industrial architectures, such as Intel's Larrabee processor [70]. To capture all temporal phase behaviour the entire parallel phase of each benchmark is run using the recommended input size. To account for variability in simulating a multi-threaded workload on a full-system simulator, it is necessary to randomise the memory access latency slightly for each data point, as described by Alameldeen and Wood [3],



(a) Preferred neighbour analysis for *fmm* and *water-nsq*.



(b) H-tree thread placement.

Figure 6.1: Thread mapping considerations: (a) shows the best neighbour lists for *fmm* and *water-nsquared*. A darker colour indicates that this core is more likely to be able to forward data to the requesting core. There is a dark region around the diagonal, which resulted in the approximate thread placement strategy shown in (b).

and run each benchmark many times to produce results with sufficient confidence. Error bars showing standard deviation are included where applicable.

6.2.2 Benchmark Selection

Due to extremely long run times when simulating a large system in an execution-driven simulator it was unfortunately necessary to reduce the number of benchmarks analysed in this part of the work. To fully explore the upside of the proposal, I decided to focus on benchmarks that showed promise during the analysis in Chapter 4. Figures 4.2 and 4.3

indicate which of the SPLASH-2 [79] and Parsec [15] benchmarks exhibit behaviours that warrant further investigation. However, I also included *ocean*, a benchmark with a low proximity hit rate of 13%, to evaluate the behaviour of Proximity Coherence with less favourable programs.

Sadly, further compromises had to be made when it was discovered that many of the Parsec benchmarks were orders of magnitude too long to simulate using academic tools, effectively excluding them from use in architectural investigations. Ultimately this lead to the execution-driven investigation using only SPLASH-2 applications.

It is hoped that Proximity Coherence would also be beneficial for Parsec benchmarks, as is implied by the ample physical locality demonstrated in Chapter 4. Further analysis is left to future work, as current tools and benchmarks have proved inadequate to fully explore the performance implications of Proximity Coherence for all applications.

6.2.3 Thread Mapping

Chapter 4 presents a detailed analysis of the interaction between thread mapping and proximity hit rate. Proximity Coherence uses this work to motivate the use of a static H-tree mapping for all benchmarks, as computing the optimal 2D mesh mappings would place additional strain on the compiler or runtime environment. Table 4.1 shows that this approximation still captures the majority of available locality. This work assumes that the conclusion still holds, despite the ISA and OS limitations described in Section 6.2.

One benefit of Proximity Coherence is that it does not require the programmer to specify architecture specific thread mappings to achieve reasonable speed ups. If it were possible to use an optimal thread mapping for each application, then Proximity Coherence would likely provide even greater benefits. However, this first investigation into

Proximity Coherence uses a simple static mapping. Investigations into the possibility of compiler or user hints are left for future work.

6.3 Experimental Results

This section evaluates Proximity Coherence in detail. A high proximity hit rate is measured for the selection of benchmarks, in line with the predicted values. As a direct consequence, the new scheme provides considerable improvements in memory access latency, which in turn improves overall program execution time. Additionally, the experiments show that in delivering these benefits, Proximity Coherence does not impose unrealistic demands on network resources. In fact, the system reduces the energy requirements of the cache hierarchy, creating a faster and more efficient coherence protocol.

Three versions of Proximity Coherence are evaluated, one implementing only *Load on S* sharing (referred to as *Prox*) and the second also providing support for *Load on E/M* sharing (referred to as *ProxF*). The third version, used to evaluate the impact of the point-to-point links, is a modified implementation of the *ProxF* protocol, where neighbouring caches are snooped via the global on-chip network (referred to as *ProxF-N*).

6.3.1 Impact on Memory Latency

Figure 6.2 shows the effects of Proximity Coherence on L1 load and store miss latencies. *Prox* achieves load latency reduction of up to 32% and 14% on average. *ProxF* provides further improvements, lowering load miss latency by an additional 2.3% on average. This results in a maximum reduction of 33% in the case of *fmm*. These improvements are obtained by avoiding unnecessary indirections to the directory, as dis-

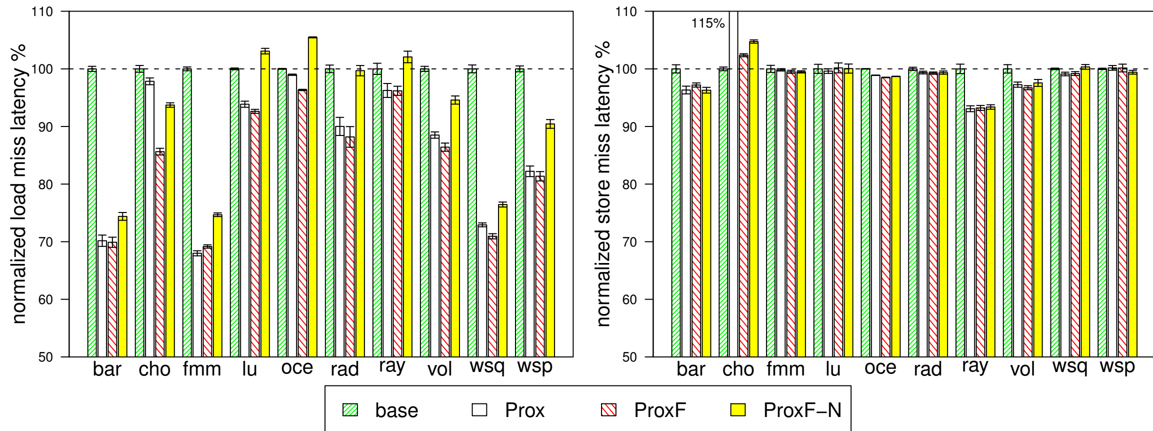


Figure 6.2: Cache miss latency reduction in % compared to a system using the MESI baseline protocol.

cussed in Section 5.3. *ProxF-N* also benefits from physical locality of shared data, but due to latencies introduced by unnecessary router traversals, there are diminished gains.

When using Proximity Coherence, store miss latencies can be marginally increased. The worst degradation in latency occurs in *cholesky* due to the serialisation of invalidations in forwarding graphs. A standard directory protocol is able to send invalidations to every sharer in parallel. In Proximity Coherence, however, some sharers can only be reached through the traversal of the forwarding graph, causing the observed increase in latency.

However, on average the *ProxF* scheme improves store miss latency by 1.4%, due to more efficiently supporting the re-acquirement of write permissions. This is particularly important in producer-consumer relationships, a common data sharing pattern. For example, should a cache line be held in state F, the core can normally re-obtain write permission with a 2-hop transaction, as described in Section 5.3.4. In *Prox* however, where no F state is implemented, a 3-hop transaction is required.

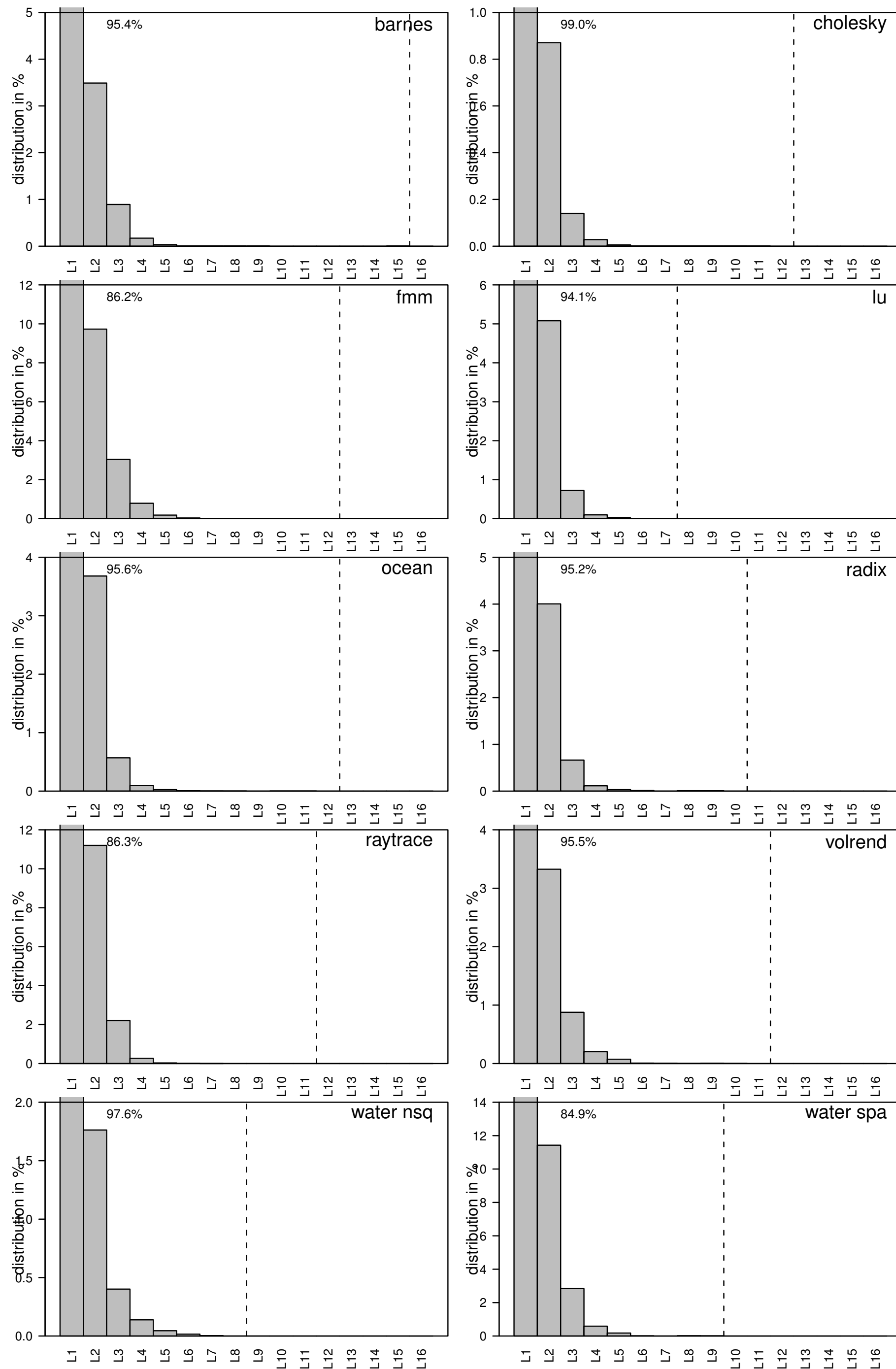


Figure 6.3: Distribution of the depth of the sharer graph at the time of an invalidation request, when using the *ProxF* scheme. The graph has in most cases only a depth of 1, resulting in negligible overhead. The vertical dashed line indicates the maximum depth observed in that program.

6.3.2 Invalidation Chain Length

In addition to the fixed overhead of checking adjacent caches, Proximity Coherence serialises invalidations within the forwarding graphs of shared data. If a forwarding graph is deep, an invalidation request will take many cycles to propagate to the end of each branch, causing slow state transitions. For Proximity Coherence to provide good performance, the depth of any forwarding graphs frequently invalidated must be low. Figure 6.3 shows the depth of invalidations encountered when using the *ProxF* scheme.

The graphs invalidated most frequently are 1 link deep, showing that data was forwarded only once before being invalidated. Over 98% of proximity invalidations are of depth less than or equal to 2. This minimises the serialisation penalty and ensures good invalidation performance for data shared through proximity hits.

6.3.3 Proximity Hit Rate

Figure 6.4 shows the measured proximity hit rates for both *Prox* and *ProxF*. For *ProxF*, *Load on S* and *Load on M* hits are shown separately.

The implementations of Proximity Coherence achieve hit rates of up to 54%, granting the latency improvements already described. The results show that in almost all cases, the measured proximity hit rate is close to the predicted values presented in Chapter 4. This is especially interesting, as the expected hit rates have been generated using an ideal thread placement, while the measured results use only an approximate placement, as described in Section 6.2. Additional variation is introduced through operating system interference. *Radix* is especially affected, as it is a particularly short running benchmark: a significantly higher proximity hit rate is observed in the full-system simulation results than the predicted value.

The results for *ProxF* show that, for each benchmark, *Load on M* forwarding

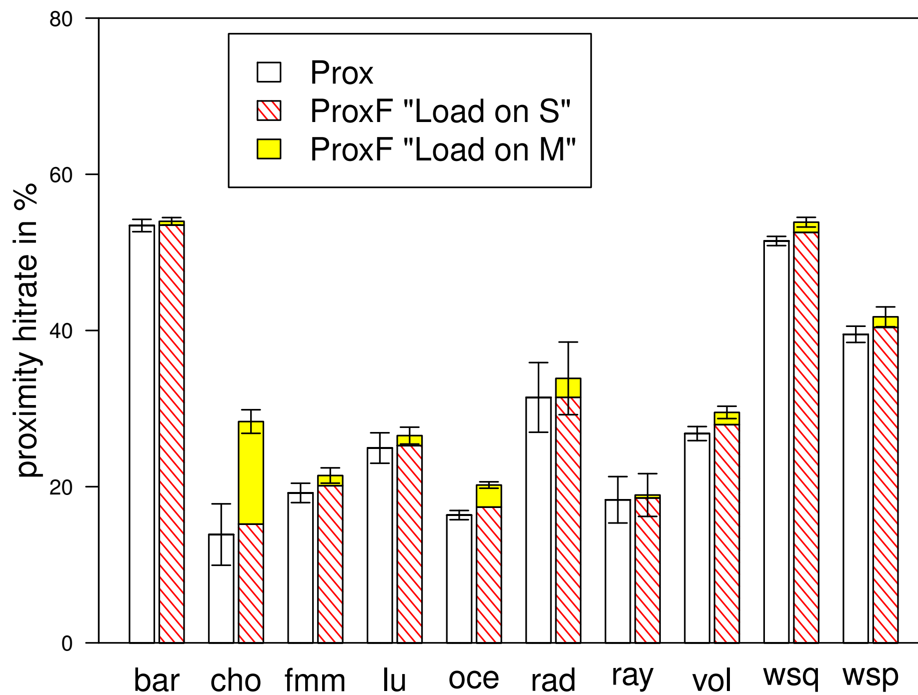


Figure 6.4: Measured proximity hit rates for *Prox* and *ProxF*.

provides only a small proportion of proximity hits, on average 2.6% and, excluding *cholesky*, just 1.4%. However, this small improvement means that more sharers are available in the system sooner and these sharers can offer data via *Load on S* forwarding, as reflected by the increased *Load on S* events for *ProxF*. This behaviour improves average proximity hit rate by an additional 3.3%. These two effects combined deliver higher than expected latency benefits, as shown in Figure 6.2. *ProxF* increases latency reduction by up to 7%, justifying the additional complexity.

6.3.4 Execution Time Improvements

Figure 6.5 shows the overall execution time improvements Proximity Coherence provides. Using the *ProxF* scheme delivers benefits of up to 13% with only *ocean* suffering a slight slow down. *Ocean* was included as an example of a program with low proximity hit

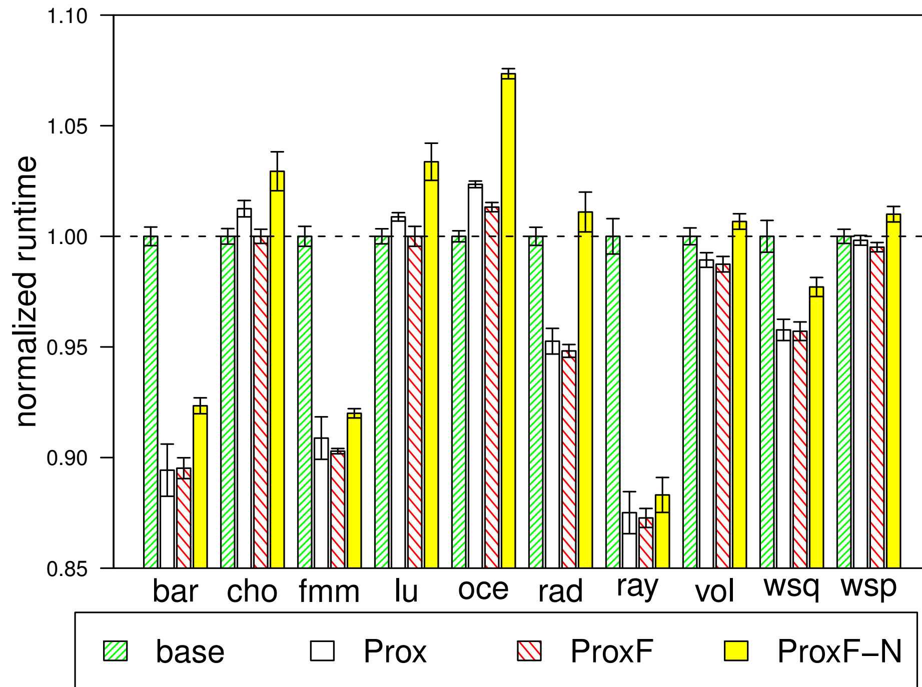


Figure 6.5: Runtime reduction compared to a system using the MESI baseline protocol. For increased clarity, the y-axis is scaled to show runtimes between 0.85x and 1.10x

rate, leading to a marginal execution time increase of 1%. Importantly however, network traffic and energy consumption are still reduced. *ProxF-N* cannot match these improvements and for six benchmarks delivers worse runtime results than the baseline system.

Although Proximity Coherence is an effective optimisation, its impact on execution time is limited by the high L1 cache hit rates observed in the chosen benchmarks. The data forwarding mechanisms of the protocol are only exercised during L1 cache misses.

6.3.5 Impact on Network Traffic

As Proximity Coherence optimises the communication in many-core systems, it is important to analyse its impact on on-chip network traffic. This study distinguishes between two types of traffic: proximity messages that are carried on the new dedicated links de-

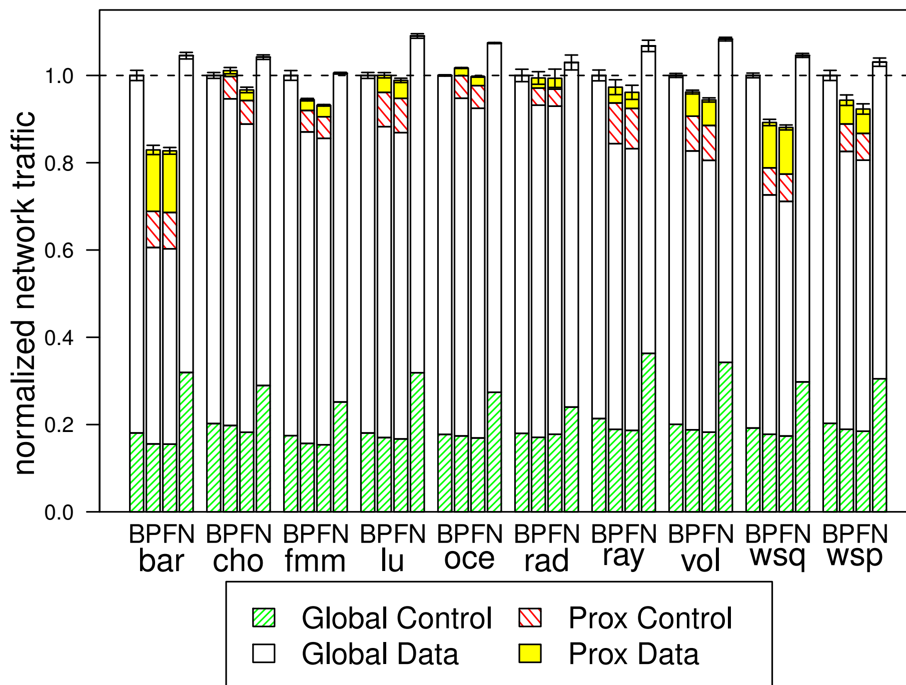


Figure 6.6: Normalised network traffic compared to a system using the MESI baseline protocol. “B” refers to the baseline system, “P” refers to *Prox*, “F” refers to *ProxF*, and “N” refers to *ProxF-N*.

scribed in Section 5.3.7 and standard messages that use the global on-chip interconnect. This distinction is necessary as the two networks have significantly different characteristics.

Figure 6.6 shows the aggregate number of bytes transferred a single hop by the on-chip network. Since all proximity messages travel on only one point-to-point link to reach their destination, they have a fixed hop count of 1. However, global network-on-chip messages may have to travel through several routers to reach their destination.

Over all benchmarks, Proximity Coherence achieves a reduction in global network-on-chip bytes transferred of between 8% and 42%. In *Prox* and *ProxF*, cache misses that would have been serviced using the global network are satisfied using the proximity network. These messages are shown on top of the standard network traffic.

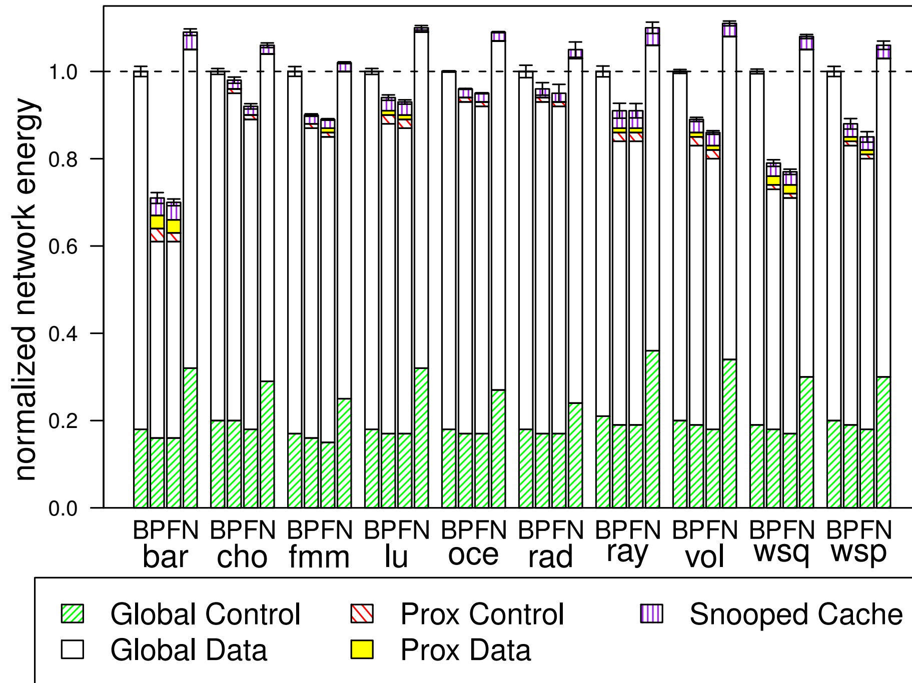


Figure 6.7: Normalised estimated network energy consumption compared to a system using the MESI baseline protocol. "B" refers to the baseline system, "P" refers to *Prox*, "F" refers to *ProxF*, and "N" referes to *ProxF-N*. Additionally shown is the energy required to perform a cache lookup in the case of a servicing a proximity request .

As discussed, *ProxF* provides several benefits over the simpler *Prox*. However, network analysis shows that these improvements create no increase in proximity link traffic. This is expected, as "Load on M" forwarding effectively turns control traffic (negative reply to a proximity request) into data traffic (positive reply). The number of requests sent and replies received remains constant.

The *ProxF-N* scheme also succeeds in reducing the amount of data traffic. However, as control messages to neighbouring cores still need to traverse two routers, the total control traffic increases to the point that it negates the savings made by reduced data traffic. For all benchmarks, *ProxF-N* generates more traffic than the baseline system, highlighting the importance of the new proximity links when implementing Proximity Coherence.

6.3.6 Impact on Energy

To confirm that Proximity Coherence is feasible to implement, the energy consumed in the two networks and the energy required for snooping the four neighbouring caches is evaluated. This study makes three assumptions. First, it is assumed that network energy consumed is proportional to the amount of data transferred. Work by Banerjee et al. [7], shows that, with effective clock-gating, this is the case. In Proximity Coherence, data messages are approximately nine times larger than control messages. As such, it is assumed that they consume nine times more energy. Second, it is assumed that when transferring a message, the energy consumed in a router is four times that which is consumed in the link. This assumption is based upon work presented by Kundu [50]. As the proximity network is composed of simple point-to-point links with no routers, it is assumed that the energy required to send a single proximity message is equal to the amount consumed by a global network link. Finally, as discussed in Section 5.2.4, it is assumed that the energy required for a single L1 cache lookup is equivalent to the amount consumed by a router processing one message. For simplicity, this study does not consider the energy saved by not performing an L2 lookup after a proximity hit.

Figure 6.7 shows the total network energy consumption under the discussed assumptions. The figure also shows the energy overhead associated with snooping caches. When using the baseline MESI protocol only 19% of energy is spent on control messages, despite their greater contribution to overall network traffic. Using either of the Proximity Coherence implementations that employ proximity links results in a reduction of between 5% and 30% in total network energy. Importantly, the reduced consumption in the global network is not nearly matched by the energy spent in the proximity links. Moreover, the total network energy saved more than offsets the additional expense of lookups in neighbouring caches. The results show that as *ProxF-N* only uses the global

on-chip interconnect, its energy requirements are up to 55% higher than *ProxF* (24% on average), further motivating the inclusion of proximity links in architectures implementing Proximity Coherence. A more detailed analysis is left to future work.

6.4 Conclusion

This chapter presents Proximity Coherence, a novel protocol that exploits the physical locality of shared data to provide efficient cache coherence in many-core architectures. The design delivers a 14% reduction in L1 load miss latency, while reducing global on-chip network traffic by 19%. For the selection of benchmarks described, Proximity Coherence achieves execution time improvements of up to 13%. The work shows that using Proximity Coherence allows network traffic and latency to be effectively traded off against additional L1 cache accesses, while simultaneously reducing energy consumed by the memory hierarchy.

These benefits emerge through the use of new dedicated links between neighbouring cores. Using these links, data is optimistically requested from adjacent cores. Coherence is then maintained through delegation of responsibility, from the directory to caches that have forwarded data. An implementation without these links is not feasible, as using the global on-chip interconnect increases the energy required by the network by 24% and reduces the obtainable latency improvements. Additionally, it is impossible to run the baseline MESI protocol transactions over the simple proximity links – such messages require more comprehensive routing, flow control and buffering. Furthermore, the resources required to form proximity links are so minimal, that reassigning their use to further increase the global-network bandwidth is not possible – increasing bandwidth requires larger crossbars and associated datapaths, not just more wires.

CHAPTER 7

Related Work

This chapter describes the similarities and differences between the two new bodies of work in this thesis – Communication Characterisation and Proximity Coherence – and the related works in their respective fields.

7.1 Communication Characterisation

The works by Woo et al. [79] and Bienia et al. [15], which present the SPLASH-2 and Parsec suites respectively, contain a large amount of information on the benchmarks used here. These characterisations focus on synchronisation overhead, size of working sets, false and true sharing, and scalability. Unlike this work, they do not evaluate temporal and spatial communication patterns, nor do they try to classify shared data access patterns.

Bienia et al. [14] also compare the SPLASH-2 and Parsec benchmark suites. However, while they examine the sharing behaviour for both suites, this data is evaluated with a particular system in mind (i.e. data sharing is only observed if the data is shared through caches). The study in this thesis focuses on sharing patterns at an address level. As such, the work presented here offers insight into the kind of communication is present in the applications, regardless of execution platform.

Chodnekar et al. [25] present a communication characterisation methodology for parallel applications. Their work focuses on temporal and spatial traffic characterisation

for a multi-node CC-NUMA system. However, their evaluation is tied to a particular physical implementation of a CC-NUMA machine. For example, the communication analysis assumes a communication network with a mesh topology. This thesis examines communication with no specific topology in mind, providing generic results for use in future work.

Hossain et al. [41] present an augmented cache coherence protocol for CMPs that tries to take advantage of producer/consumer and migratory sharing. The protocol uses heuristics and additional status bits in each cache line to identify these patterns dynamically with local information available at each L1. All traffic observed in the system is then characterised using these heuristics. In contrast, the communication characterisation presented in this thesis uses global knowledge about the application and does not miss patterns masked due to conflict misses. Additionally, their communication evaluation only includes a selection of programs from the SPLASH-1/2 benchmark suites; the evaluation in Chapter 3 of this research also considers Parsec benchmarks.

There are many other publications that augment the cache coherence protocol to take advantage of specific sharing pattern such as [23; 72]. Many such works target multi-node systems. Similar to Hossain's work, they use a heuristic and only present communication properties of applications that exhibit improved performance with their scheme. The evaluation in Chapter 3 of this research considers all SPLASH-2 applications and also the emerging workloads in the Parsec suite. None of these studies investigate how much traffic falls into a particular category.

7.2 Proximity Coherence

To the best of my knowledge, this work is the first to suggest the use of dedicated wires to snoop neighbouring caches in a many-core processor. However, prior work exists

that tries to exploit proximity in a chip-multiprocessor or takes into consideration the special properties of chip-multiprocessors as opposed to multi-node systems.

Cheng et al. [24] optimise the energy demand of the on-chip interconnect by providing different networks for different coherence message types. Unlike the Proximity Coherence scheme, they do not explore the new opportunities of a many-core design and focus solely on optimising the on-chip network for an existing cache coherence protocol.

Brown et al. [17] describe an augmentation to the coherence mechanism that takes into account the proximity of available sharers when the directory serves an L1 cache miss and cannot provide a copy from its L2 cache bank. Unlike the Proximity Coherence scheme, Brown's scheme does not avoid the extra hop to the directory and cannot utilise an inexpensive point-to-point network that provides a copy from a neighbouring sharer. Finally, the proposed changes are orthogonal to Proximity Coherence and combining both schemes may be beneficial.

Eisley et al. [31] propose a coherence mechanism that is directly embedded into the interconnection network routers. The mechanism works by building tree structures in the network routers that redirect requests to the directory towards a nearby sharer, if the request happens to traverse a node that is part of the tree. However, depending on the routing, it is entirely possible that the request will miss an adjacent sharer and proceed across the network. Proximity Coherence will always probe neighbouring tiles, and is guaranteed to find adjacent copies. Furthermore, the scheme increases the processing time of the router, dealing with both routing and coherence protocol tasks. Finally, the work does not present execution time statistics, which prevents any direct comparison of performance.

Enright Jerger et al. [45] propose a protocol that uses a tree structure to maintain coherence across several sharers. The root of the tree acts as an ordering point for requests. While their scheme uses a coarse-grained coherence mechanism, Proximity Coherence

maintains coherence at cache line granularity. In addition, their scheme also results in an increase of global network traffic by a factor of two to three over a standard directory protocol, drastically reducing the efficiency of the proposal. In contrast, Proximity Coherence delivers improved performance and reduces energy consumption.

Hossain et al. [41] present a scheme in which an L1 cache also sends a request to a neighbouring cache instead of sending a request to the directory. However, since they use the global on-chip network for such requests, rather than the novel dedicated links used in this work, their definition of neighbouring is a more relaxed "close-by" instead of adjacent. Furthermore, while the data is provided by this "close-by" cache, the directory functions are not delegated to this cache. Instead, the directory is immediately informed and the provided data can only be used once the directory has acknowledged the forwarding. The main performance gain in their system comes from control messages having a lower latency than data messages. Proximity Coherence assumes a global network that delivers data and control messages with the same latency. Additionally this thesis models state of the art router latencies [7]. As detailed in Hossain's work, using such a low latency network reduces the benefits gained through their scheme. Finally, the Proximity Coherence protocol delegates coherence responsibility to the L1 that forwarded the data, such that the data is usable immediately; an acknowledgement from the directory is not needed.

Cache coherence protocols have been proposed that use linked-lists to track sharers in a multi-processor system [43; 62]. Although sharers are also tracked using pointers in Proximity Coherence, the scheme differs significantly: it tracks sharers in an acyclic graph and takes physical locality information into account. Further differences are found due to the proximity-link network introduced by this work.

Cheng et al. [23] propose a scheme that delegates directory responsibilities to other nodes in the system. The goal is to transform 3-hop transactions into 2-hop transac-

tions. However, their design is optimised for a multi-node system and unlike Proximity Coherence, the delegations only happen after a stable producer-consumer relationship has been detected. The Proximity Coherence scheme uses an optimistic mechanism and establishes delegation immediately.

Ros et al. [69] propose a cache coherence protocol for tiled CMPs. Similar to the work by Cheng et al., the scheme aims to avoid long latency 3-hop transitions by delegating the directory responsibility to the owner node. While the protocol considers the limited storage requirements in a CMP system, it does not take advantage of the opportunities offered by the low latency on-chip interconnect. Implementing this scheme in a multi-node system may obtain similar improvements.

Kaxiras et al. [48] present work which evaluates the accuracy of a variety of coherence prediction schemes. These mechanisms are used to dynamically predict remote nodes to forward newly written data to. Although the predictors can achieve good accuracy for the small selection of benchmarks evaluated, such schemes are not well suited to a system in which network accesses have a sizeable energy cost. Proximity Coherence employs a pull mechanism, ensuring any data moved will be consumed. In comparison, coherence predictors can move large data packets that will never be consumed. A failed proximity snoop will have considerably lower energy penalty than a mispredicted data packet forwarded by a coherence predictor. Furthermore, Proximity Coherence makes efficient use of existing hardware mechanisms, without the need for extra prediction tables.

Lai et al. [51] propose another coherence predictor scheme that use tables containing a history of previous protocol operations to speculatively send read requests to remote nodes. Similar to work by Kaxiras, the predictors achieve good accuracy for the small subset of benchmarks analysed, but performance gains are at the cost of additional hardware structures to track predictor state. Proximity Coherence does not require

any such structures and is specifically designed to exploit physical locality in order to improve energy efficiency.

Lenoski et al. [54] present the Dash architecture in which computation nodes are arranged in multiple clusters, and these clusters are connected by a directory protocol. Each cluster runs a private snooping protocol to facilitate fast inter-cluster sharing. This two-level protocol is similar to the scheme proposed in this work. In Proximity Coherence the first level of the protocol is maintained by the sharing graphs of data forwarded over the local links. The second level is a directory protocol, however in Proximity Coherence, all nodes are clients of the directory. The advantage of Proximity Coherence comes from the potential for data to be shared over proximity links to a large number of nodes. In Dash, data can only be efficiently shared within the strict set of local nodes connected in the cluster. In a modern chip-multiprocessor this causes a non-uniform cost to accessing physically local data. The required data may be available in an adjacent tile, however if the tile is part of a different cluster then it is necessary to process the access via the slower directory protocol. Of course Dash was designed for multi-node systems, in which such problems are not of concern, but they are a good example of the draw backs of implementing multi-node protocols on a chip-multiprocessor.

Li et al. [56] propose a system that tracks coherence at a page level, and employs a simple predictor to direct requests to the likely owner of a page. The scheme is well suited to programs that exhibit coarse grained sharing, but would likely struggle with more recent benchmarks using finer grained communication. Proximity Coherence combats this by tracking coherency at a cache line granularity in both the proximity link, and directory level protocols. The prediction in Li's scheme is based on tracking a probable owner of each page in the system in each local page table. When a node requires access to a page it first sends a request to the probable owner. If the probable owner does not have ownership of the page, it will in turn send the request to its own probably owner

for the page. Eventually the request will find the owner of the page, and the relevant permissions will be returned to the requesting node. This technique is viable when the movement of pages between nodes is rare. However if such a technique were used at a cache line granularity below the L1, as in Proximity Coherence, the extra latency attached to each access would likely negate any benefit of a successful prediction. Again, similar to the Dash architecture, Li's work was proposed for multi-node systems. In chip-multiprocessors it is necessary to employ different techniques to provide efficient coherence.

CHAPTER 8

Conclusions

8.1 Thesis Summary

The advent of chip-multiprocessors as the dominant processor architecture has introduced new design constraints relating to the cost of both communication and computation. Previous designs used in processor architecture can be revisited, and optimised specifically for these highly integrated parallel architectures.

The majority of architectures proposed by both academic and industry researchers are now tiled designs featuring substantial and complex network hardware to connect the large number of cores. At the same time as increasing the number of parallel compute units, designers often aim to maintain a shared memory programming model for the sake of both compatibility and programmability. Achieving this requires carefully designed coherence protocols to support this abstraction on increasingly distributed compute fabrics, and when implementing these protocols on new chip-multiprocessors, additional design optimisations present themselves.

The first step in exploring these opportunities is to thoroughly analyse the behaviour of the applications to be run on the new architectures. Chapter 3 presents an analysis of the two most prevalent parallel benchmark suites — SPLASH-2 and PARSEC. A trace-driven simulation infrastructure was developed to allow the investigation of communication patterns of the benchmarks, leading to the observation of migratory, read-only and producer-consumer sharing patterns. The temporal and spatial characteristics of

communicating memory accesses are measured and the implications of the results on the design of coherence protocols, network-on-chip architectures, and thread mapping are considered. Crucially, it is discovered that there is considerable, and well structured, locality in communicating accesses when ordering threads by the OS-assigned thread number.

Chapter 4 summarises this new, inter-processor locality and describes the importance of this discovery for the design of future chip-multiprocessor communication systems. The impact of thread mapping is analysed, and shows that a simple mapping can capture the majority of potential the locality.

Chapter 5 investigates the implications of the locality of shared data and describes how memory accesses satisfied by physically local private memory can offer significantly more efficient coherent operation. A novel coherence protocol – Proximity Coherence – is proposed, in which global network traversals are avoided through the use of additional cache look ups in adjacent local nodes, hence exploiting the locality of shared data observed in Chapter 3. Proximity Links – low cost wiring between adjacent tiles – are introduced and are used for a subset of coherence messages. Specialising the network in this way allows for dramatic increases in communication efficiency. The mechanism of coherence delegation and data forwarding under the new scheme is described, as well as the state machine design. The techniques used to avoid protocol races are listed, explaining how Proximity Coherence achieves resilience under heavy workloads.

Chapter 6 presents the evaluation of the Proximity Coherence protocol, measuring the performance and energy benefits of exploiting physical locality of shared data. The entire protocol was implemented using the GEMS simulation environment to allow the execution time of each benchmark to be evaluated. Network traffic characteristics are measured, showing a reduction in utilisation of the high power global network-on-chip. Finally the energy costs of running Proximity Coherence are estimated, suggesting that

exploiting physical locality of shared data with a protocol running over a carefully designed communication fabric can not only improve performance, but also reduce energy consumption.

This work confirms that detailed analysis of communication behaviour of benchmarks running on novel chip-multiprocessor architectures motivates the extension of existing coherence protocols to fully exploit the shifting costs of communication and computation found in new processors. Proximity Coherence is designed in this way, leveraging the physical layout of the CMP, combined with benchmark behaviour, to provide efficient caching of shared data.

To summarise, this thesis has produced the following contributions to the field:

- Comprehensive analysis of communication patterns in both legacy and emerging shared-memory applications.
- Discovery of physical locality in many parallel benchmark applications.
- Proposal of low-cost links between physically local tiles to be used specifically to exploit this new locality.
- Design and evaluation of Proximity Coherence, a new protocol to use the low-cost local links to improve performance and reduce energy consumption of shared-memory chip-multiprocessors.

8.2 Future Directions

This thesis presents work that provides an excellent starting point for a number of exciting avenues of further investigation. Both the research of communication analysis in Chapter 3, and the design and evaluation of Proximity Coherence in chapters 5 and 6, present several opportunities.

8.2.1 Communication Characterisation

The tools developed for the communication characterisation work presented in Chapter 3 can be used in a number of ways for further research. In particular the simple code structure and fast runtimes make the simulator ideal for the early evaluation of ideas during the infancy of research projects. The tools have already been used by researchers to investigate on-chip optical interconnect parameters [67]. Furthermore, when an optimisation opportunity presents itself, the trace driven results can suggest particular benchmarks that would benefit from more detailed simulation. This can save considerable time when dealing with slower, event-driven simulation architectures. Beyond the infrastructure, the results already gathered will prove useful in directing future research efforts. The temporal and spatial locality of communicating accesses could be used for the validation of synthetic traffic patterns in network-on-chip design, or to provide initial results for research into optimal thread-mappings of multithreaded benchmarks to CMPs.

8.2.2 Proximity Coherence

In addition to the evaluation I have presented in this work, it is important to consider the impact of Proximity Coherence across a wider selection of benchmarks. With more mature simulation tools, and suitable benchmark suites it will be possible to make broader conclusions about the benefits and limitations of the scheme. Another important aspect to evaluating Proximity Coherence is a sensitivity analysis to the many parameters found in chip-multiprocessor systems. This work uses what can be considered typical values for each parameter, but further conclusions about the feasibility of Proximity Coherence could be drawn if it was found the scheme worked particularly well for certain configurations.

Looking forward, Proximity Coherence also presents many opportunities for additional research. First, reducing the number of unsuccessful cache snoops by using dynamic prediction may be possible. Also of interest is the potential benefit of an OS-based scheme to disable Proximity Coherence in situations where it is either not required, or has detrimental effects on performance. Such a scheme would require simply changing a single state transition, disabling snooping. Second, Proximity Coherence could be implemented on a strictly non-inclusive cache hierarchy that maximises on-chip storage utilisation. Third, it is likely that a processor architecture employing chip-stacking would allow for a greater number of proximity-links to be added, further improving the chances of delivering a proximity hit. Finally, restructuring the benchmark algorithms could increase the physical locality of shared data, improving the proximity hit rate. In such a scheme, Proximity Coherence would provide efficient support for message-passing style communication between physically local cores, while still supporting a fallback of a fully coherent shared-memory. Optimising communication then becomes an optional performance layer, offering an interesting new platform for software and hardware engineers alike.

Bibliography

[1] A.Agarwal, L.Bao, J.Brown, B.Edwards, M.Mattina, C.Miao, C.Ramey, and D.Wentzlaff. Tile Processor: Embedded Multicore for Networking and Multimedia. In *Hot Chips 19*, Aug. 2007.

[2] V. Agarwal, M. S. Hrishikesh, S. W. Keckler, and D. Burger. Clock Rate Versus IPC: The End of the Road for Conventional Microarchitectures. In *Proceedings of the 27th Annual International Symposium on Computer Architecture*, ISCA '00, pages 248–259, New York, NY, USA, 2000. ACM.

[3] A. R. Alameldeen and D. A. Wood. Variability in Architectural Simulations of Multi-threaded Workloads. In *Proceedings of the 9th International Symposium on High-Performance Computer Architecture*, pages 7–18, Feb. 2003.

[4] K. Asanovic, R. Bodik, B. C. Catanzaro, J. J. Gebis, P. Husbands, K. Keutzer, D. A. Patterson, W. L. Plishker, J. Shalf, S. W. Williams, and K. A. Yelick. The Landscape of Parallel Computing Research: A View from Berkeley. Technical Report UCB/EECS-2006-183, EECS Department, University of California, Berkeley, Dec 2006.

[5] M. Azimi, N. Cherukuri, D. N. Jayasimha, A. Kumar, P. Kundu, J. Park, and I. S.

A. S. Vaidya. Integration Challenges and Tradeoffs for Tera-scale Architectures. *Intel Technology Journal*, 11(3), August 2007.

[6] F. Baker. Requirements for IP Version 4 Routers. In *RFC 1812*, United States, 1995.

[7] A. Banerjee, P. T. Wolkotte, R. D. Mullins, S. W. Moore, and G. J. Smit. An Energy and Performance Exploration of Network-on-Chip Architectures. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, 17(3):319–329, Mar. 2009.

[8] G. Barnes. A Method for Implementing Lock-Free Shared-Data Structures. In *Proceedings of the Fifth Annual ACM Symposium on Parallel Algorithms and Architectures*, SPAA '93, pages 261–270, New York, NY, USA, 1993. ACM.

[9] N. Barrow-Williams, C. Fensch, and S. Moore. A Communication Characterisation of Splash-2 and Parsec. *IEEE Workload Characterization Symposium*, 0:86–97, 2009.

[10] N. Barrow-Williams, C. Fensch, and S. Moore. Proximity Coherence for Chip Multiprocessors. In *Parallel Architectures and Complication Techniques*, Sept. 2010.

[11] S. Bell, B. Edwards, J. Amann, R. Conlin, K. Joyce, V. Leung, J. MacKay, M. Reif, L. Bao, J. Brown, M. Mattina, C.-C. Miao, C. Ramey, D. Wentzlaff, W. Anderson, E. Berger, N. Fairbanks, D. Khan, F. Montenegro, J. Stickney, and J. Zook. TILE64 - Processor: A 64-Core SoC with Mesh Interconnect. In *Solid-State Circuits Conference, 2008. ISSCC 2008. Digest of Technical Papers. IEEE International*, pages 88 –598, Feb. 2008.

[12] J. K. Bennett, J. B. Carter, and W. Zwaenepoel. Adaptive Software Cache Management for Distributed Shared Memory Architectures. In *Proceedings of the 17th*

Annual International Symposium on Computer Architecture, pages 125–134, June 1990.

[13] R. A. Bergamaschi and W. R. Lee. Designing Systems-on-Chip Using Cores. In *Proceedings of the 37th Annual Design Automation Conference*, DAC '00, pages 420–425, New York, NY, USA, 2000. ACM.

[14] C. Bienia, S. Kumar, and K. Li. PARSEC vs. SPLASH-2: A Quantitative Comparison of Two Multithreaded Benchmark Suites on Chip-Multiprocessors. In *Proceedings of IEEE International Symposium on Workload Characterization*, pages 47–56, Sept. 2008.

[15] C. Bienia, S. Kumar, J. P. Singh, and K. Li. The PARSEC Benchmark Suite: Characterization and Architectural Implications. In *Proceedings of the 17th International Conference on Parallel Architectures and Compilation Techniques*, pages 72–81, Oct. 2008.

[16] C. Bienia and K. Li. PARSEC 2.0: A New Benchmark Suite for Chip-Multiprocessors. In *Proceedings of the 5th Annual Workshop on Modelling, Benchmarking and Simulation*, June 2009.

[17] J. A. Brown, R. Kumar, and D. Tullsen. Proximity-Aware Directory-based Coherence for Multi-Core Processor Architectures. In *Proceedings of SPAA 19*, pages 126–134, June 2007.

[18] J. F. Cantin, M. H. Lipasti, and J. E. Smith. Improving Multiprocessor Performance with Coarse-Grain Coherence Tracking. In *Proceedings of the 32nd annual international symposium on Computer Architecture*, ISCA '05, pages 246–257, Washington, DC, USA, 2005. IEEE Computer Society.

[19] J. G. Castanos, L. Ceze, K. Strauss, and H. S. Warren Jr. Evaluation of a Multithreaded Architecture for Cellular Computing. In *Proceedings of the 8th International Symposium on High-Performance Computer Architecture*, HPCA '02, pages 311–, Washington, DC, USA, 2002. IEEE Computer Society.

[20] R. Chandra, L. Dagum, D. Kohr, D. Maydan, J. McDonald, and R. Menon. *Parallel programming in OpenMP*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2001.

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

[22] L. Cheng and J. B. Carter. Extending CC-NUMA Systems to Support Write Update Optimizations. In *Proceedings of the 2008 ACM/IEEE conference on Supercomputing*, SC '08, pages 30:1–30:12, Piscataway, NJ, USA, 2008. IEEE Press.

[23] L. Cheng, J. B. Carter, and D. Dai. An Adaptive Cache Coherence Protocol Optimized for Producer-Consumer Sharing. In *Proceedings of the 13rd International Symposium on High-Performance Computer Architecture*, pages 328–339, Feb. 2007.

[24] L. Cheng, N. Muralimanohar, K. Ramani, R. Balasubramonian, and J. B. Carter. Interconnect-Aware Coherence Protocols for Chip Multiprocessors. In *In Proceedings of ISCA-33*, June 2006.

[25] S. Chodnekar, V. Srinivasan, A. S. Vaidya, A. Sivasubramaniam, and C. R. Das. Towards a Communication Characterization Methodology for Parallel Applications. In *Proceedings of the 3rd International Symposium on High-Performance Computer Architecture*, pages 310–319, Feb. 1997.

[26] A. L. Cox and R. J. Fowler. Adaptive Cache Coherency for Detecting Migratory Shared Data. In *Proceedings of the 20th Annual International Symposium on Computer Architecture*, ISCA '93, pages 98–108, New York, NY, USA, 1993. ACM.

[27] D. Culler, J. Singh, and A. Gupta. *Parallel Computer Architecture: A Hardware/Software Approach*. Morgan Kaufmann, 1st edition, 1998. The Morgan Kaufmann Series in Computer Architecture and Design.

[28] W. J. Dally and B. Towles. Route Packets, Not Wires: On-Chip Interconnection Networks. In *Proceedings of the 38th annual Design Automation Conference*, DAC '01, pages 684–689, New York, NY, USA, 2001. ACM.

[29] D. L. Dill, A. J. Drexler, A. J. Hu, and C. H. Yang. Protocol Verification as a Hardware Design Aid. In *Proceedings of the 1991 IEEE International Conference on Computer Design on VLSI in Computer & Processors*, ICCD '92, pages 522–525, Washington, DC, USA, 1992. IEEE Computer Society.

[30] M. Dubois, C. Scheurich, and F. Briggs. Memory Access Buffering in Multiprocessors. In *Proceedings of the 13th annual International Symposium on Computer Architecture*, ISCA '86, pages 434–442, Los Alamitos, CA, USA, 1986. IEEE Computer Society Press.

[31] N. Eisley, L.-S. Peh, and L. Shang. In-Network Cache Coherence. In *Proceedings of MICRO 39*, pages 321–332, Dec. 2006.

[32] J. Emer, M. D. Hill, Y. N. Patt, J. J. Yi, D. Chiou, and R. Sendag. Single-Threaded vs. Multithreaded: Where Should We Focus? *IEEE Micro*, 27:14–24, November 2007.

[33] C. Fensch and M. Cintra. An OS-based Alternative to Full Hardware Coherence on Tiled CMPs. In *HPCA*, pages 355–366. IEEE Computer Society, 2008.

[34] D. Flynn. AMBA: Enabling Reusable On-Chip Designs. *IEEE Micro*, 17:20–27, July 1997.

[35] K. Gharachorloo, D. Lenoski, J. Laudon, P. Gibbons, A. Gupta, and J. Hennessy. Memory Consistency and Event Ordering in Scalable Shared-Memory Multiprocessors. In *Proceedings of the 17th Annual International Symposium on Computer Architecture*, ISCA '90, pages 15–26, New York, NY, USA, 1990. ACM.

[36] J. Gosling, B. Joy, and G. L. Steele. *The Java Language Specification*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1st edition, 1996.

[37] A. Gupta and W.-D. Weber. Cache Invalidation Patterns in Shared-Memory Multiprocessors. *IEEE Trans. Comput.*, 41:794–810, July 1992.

[38] M. Herlihy and J. E. B. Moss. Transactional Memory: Architectural Support for Lock-Free Data Structures. In *Proceedings of the 20th Annual International Symposium on Computer Architecture*, ISCA '93, pages 289–300, New York, NY, USA, 1993. ACM.

[39] M. D. Hill. Multiprocessors Should Support Simple Memory-Consistency Models. *Computer*, 31:28–34, August 1998.

[40] M. D. Hill and M. R. Marty. Amdahl's Law in the Multicore Era. *Computer*, 41:33–38, July 2008.

[41] H. Hossain, S. Dwarkadas, and M. C. Huang. Improving Support for Locality and Fine-Grain Sharing in Chip Multiprocessors. In *Proceedings of the 17th Inter-*

national Conference on Parallel Architectures and Compilation Techniques, pages 155–165, Oct. 2008.

[42] ITRS Committee. International Technology Roadmap for Semiconductors, 2010. `http://public.itrs.net/`.

[43] D. V. James, A. T. Laundrie, S. Gjessing, and G. Sohi. Distributed-directory scheme: Scalable Coherent Interface. *IEEE Computer*, 23(6):74–77, 1990.

[44] N. D. E. Jerger, L.-S. Peh, and M. H. Lipasti. Circuit-Switched Coherence. In *Proceedings of the Second ACM/IEEE International Symposium on Networks-on-Chip*, NOCS '08, pages 193–202, Washington, DC, USA, 2008. IEEE Computer Society.

[45] N. D. E. Jerger, L.-S. Peh, and M. H. Lipasti. Virtual Tree Coherence: Leveraging Regions and In-network Multicast Trees for Scalable Cache Coherence. In *Proceedings of MICRO 41*, pages 35–46, Nov. 2008.

[46] N. E. Jerger, L.-S. Peh, and M. Lipasti. Virtual Circuit Tree Multicasting: A Case for On-Chip Hardware Multicast Support. In *Proceedings of the 35th Annual International Symposium on Computer Architecture*, ISCA '08, pages 229–240, Washington, DC, USA, 2008. IEEE Computer Society.

[47] C. R. Johns and D. A. Brokenshire. Introduction to the Cell Broadband Engine Architecture. *IBM Journal of Research and Development*, 51(5):503–520, 2007.

[48] S. Kaxiras and C. Young. Coherence Communication Prediction in Shared-Memory Multiprocessors. In *HPCA*, pages 156–167, 2000.

[49] T. Kgil, A. Saidi, N. Binkert, S. Reinhardt, K. Flautner, and T. Mudge. PicoServer:

Using 3D Stacking Technology to Build Energy Efficient Servers. *J. Emerg. Technol. Comput. Syst.*, 4:16:1–16:34, November 2008.

[50] P. Kundu. On-Die Interconnects for next generation CMPs. *Presentation at Workshop on On- and Off-Chip Interconnection Networks for Multicore Systems*, Dec. 2006.

[51] A. Lai and B. Falsafi. Memory Sharing Predictor: The Key to a Speculative Coherent DSM. In *In Proceedings of the 26th annual International Symposium on Computer Architecture*, pages 172–183, 1999.

[52] L. Lamport. How to Make a Multiprocessor Computer That Correctly Executes Multiprocess Programs. *IEEE Trans. Comput.*, 28:690–691, September 1979.

[53] J. Laudon and L. Spracklen. The Coming Wave of Multithreaded Chip Multiprocessors. *International Journal of Parallel Programming*, 35(3):299–330, 2007.

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

[55] A. S. Leon, J. L. Shin, K. W. Tam, W. Bryg, F. Schumacher, P. Kongetira, D. Weisner, and A. Strong. A Power-Efficient High-Throughput 32-Thread SPARC Processor. In *Digest of Technical Papers of the International Solid-State Circuits Conference (ISSCC)*, volume 2, pages 2–4, Feb. 2006.

[56] K. Li and P. Hudak. Memory Coherence in Shared Virtual Memory Systems. *ACM Trans. Comput. Syst.*, 7:321–359, November 1989.

[57] P. S. Magnusson, M. Christensson, J. Eskilson, D. Forsgren, G. Hllberg, J. Hgberg,

F. Larsson, A. Moestedt, and B. Werner. Simics: A Full System Simulation Platform. *IEEE Computer*, 35(2):50–58, Feb. 2002.

[58] M. M. K. Martin, D. J. Sorin, B. M. Beckmann, M. R. Marty, M. Xu, A. R. Alameldeen, K. E. Moore, M. D. Hill, and D. A. Wood. Multifacet's General Execution-driven Multiprocessor Simulator (GEMS) Toolset. *SIGARCH Computer Architecture News*, 33(4):92–99, Nov. 2005.

[59] G. E. Moore. Cramming More Components onto Integrated Circuits. *Electronics*, 38(8), April 1965.

[60] S. Moore and D. Greenfield. The Next Resource War: Computation vs. Communication. In *Proceedings of the 2008 International Workshop on System Level Interconnect Prediction*, SLIP '08, pages 81–86, New York, NY, USA, 2008. ACM.

[61] R. Mullins, A. West, and S. Moore. Low-Latency Virtual-Channel Routers for On-Chip Networks. In *Proceedings of the 31st Annual International Symposium on Computer Architecture*, ISCA '04, pages 188–, Washington, DC, USA, 2004. IEEE Computer Society.

[62] A. Nowatzyk, G. Aybay, M. C. Browne, E. J. Kelly, M. Parkin, B. Radke, and S. Vishin. The S3.mp Scalable Shared Memory Multiprocessor. In *Proceedings of the 1995 International Conference on Parallel Processing*, volume 1, pages 1–10, Aug. 1995.

[63] K. Olukotun, B. A. Nayfeh, L. Hammond, K. Wilson, and K. Chang. The Case for a Single-Chip Multiprocessor. In *Proceedings of the Seventh International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS-VII, pages 2–11, New York, NY, USA, 1996. ACM.

[64] S. Palacharla, N. P. Jouppi, and J. E. Smith. Complexity-Effective Superscalar Processors. In *Proceedings of the 24th Annual International Symposium on Computer Architecture*, ISCA '97, pages 206–218, New York, NY, USA, 1997. ACM.

[65] M. S. Papamarcos and J. H. Patel. A Low-Overhead Coherence Solution for Multiprocessors with Private Cache Memories. In *Proceedings of the 11th annual International Symposium on Computer Architecture*, pages 348–354, June 1984.

[66] L.-S. Peh and W. J. Dally. A Delay Model and Speculative Architecture for Pipelined Routers. In *Proceedings of the 7th International Symposium on High-Performance Computer Architecture*, HPCA '01, pages 255–, Washington, DC, USA, 2001. IEEE Computer Society.

[67] S. M. Philip Watts, Nick Barrow-Williams. Requirements of Low Power Photonic Networks for Distributed Shared Memory Computers. In *Optical Fiber Communication Conference and Exposition*, March 2011.

[68] J. Postel. User Datagram Protocol. In *RFC 768*, United States, 1980.

[69] A. Ros, M. E. Acacio, and J. M. Garca. DiCo-CMP: Efficient Cache Coherency in Tiled CMP Architectures. In *IEEE International Symposium on Parallel and Distributed Processing*, pages 1–11, Apr. 2008.

[70] L. Seiler, D. Carmean, E. Sprangle, T. Forsyth, M. Abrash, P. Dubey, S. Junkins, A. Lake, J. Sugerman, R. Cavin, R. Espasa, E. Grochowski, T. Juan, and P. Hanrahan. Larrabee: A Many-Core x86 Architecture for Visual Computing. *ACM Trans. Graph.*, 27(3):1–15, Aug. 2008.

[71] J. P. Singh, W.-D. Weber, and A. Gupta. SPLASH: Stanford Parallel Applications for Shared-Memory. *SIGARCH Computer Architecture News*, 20:5–44, March 1992.

[72] P. Stenstrm, M. Brorsson, and L. Sandberg. An Adaptive Cache Coherence Protocol Optimized for Migratory Sharing. In *Proceedings of the 20th Annual International Symposium on Computer Architecture*, pages 109–118, May 1993.

[73] M. B. Taylor, W. Lee, J. E. Miller, D. Wentzlaff, I. Bratt, B. Greenwald, H. Hoffmann, P. Johnson, J. S. Kim, J. Psota, A. Saraf, N. Shnidman, V. Strumpen, M. Frank, S. P. Amarasinghe, and A. Agarwal. Evaluation of the Raw Microprocessor: An Exposed-Wire-Delay Architecture for ILP and Streams. In *ISCA*, pages 2–13. IEEE Computer Society, 2004.

[74] S. Thoziyoor, N. Muralimanohar, J. H. Ahn, and N. P. Jouppi. Cacti 5.1. Technical Report HPL-2008-20, HP Labs, 2008.

[75] S. Vangal, J. Howard, G. Ruhl, S. Dighe, H. Wilson, J. Tschanz, D. Finan, A. Singh, T. Jacob, S. Jain, V. Erraguntla, C. Roberts, Y. Hoskote, N. Borkar, and S. Borkar. An 80-Tile Sub-100-W TeraFLOPS Processor in 65-nm CMOS. *Solid-State Circuits, IEEE Journal of*, 43(1):29–41, Jan. 2008.

[76] D. W. Wall. Limits of Instruction-Level Parallelism. In *Proceedings of the Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS-IV, pages 176–188, New York, NY, USA, 1991. ACM.

[77] W.-D. Weber and A. Gupta. Analysis of Cache Invalidation Patterns in Multiprocessors. In *Proceedings of the 3rd International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 243–256, Apr. 1989.

[78] D. Wentzlaff, P. Griffin, H. Hoffmann, L. Bao, B. Edwards, C. Ramey, M. Mattina,

C.-C. Miao, J. Brown, and A. Agarwal. On-Chip Interconnection Architecture of the Tile Processor. *Micro, IEEE*, 27(5):15 –31, 2007.

[79] S. C. Woo, M. Ohara, E. Torrie, J. P. Singh, and A. Gupta. The SPLASH-2 Programs: Characterization and Methodological Considerations. In *Proceedings of the 22nd Annual International Symposium on Computer Architecture*, pages 24–36, June 1995.

[80] J. Zebchuk, V. Srinivasan, M. K. Qureshi, and A. Moshovos. A Tagless Coherence Directory. In *Proceedings of the 42nd Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO 42, pages 423–434, New York, NY, USA, 2009. ACM.

[81] D. Ziakas, A. Baum, R. A. Maddox, and R. J. Safranek. QuickPath Interconnect Architectural Features Supporting Scalable System Architectures. In *Proceedings of the 2010 18th IEEE Symposium on High Performance Interconnects*, HOTI '10, pages 1–6, Washington, DC, USA, 2010. IEEE Computer Society.