An Efficient Cache Coherence Mechanism for Chip Multiprocessors

by Abdullah Kayi

B.S. Degree in Electrical and Electronics Engineering, June 2001
Middle East Technical University, Ankara, Türkiye

M.S. Degree in Computer Science, December 2003
University of Arkansas at Little Rock, USA

A Dissertation submitted to

The Faculty of
The School of Engineering and Applied Science of
the George Washington University in partial satisfaction
of the requirements for the degree of
Doctor of Philosophy

August 31, 2011

Dissertation directed by
Tarek El-Ghazawi
Professor of Engineering and Applied Science

The School of Engineering and Applied Science of The George Washington University certifies that Abdullah Kayi has passed the Final Examination for the degree of Doctor of Philosophy as of June 17, 2011. This is the final and approved form of the dissertation.

An Efficient Cache Coherence Mechanism for Chip Multiprocessors

Abdullah Kayi

Dissertation Research Committee:

Tarek El-Ghazawi, Professor of Engineering and Applied Science,

Dissertation Director

Edward Della Torre, Professor of Engineering and Applied Science,

Committee Member

Suresh Subramaniam, Professor of Engineering and Applied Science,

Committee Member

Abdou Youssef, Professor of Engineering and Applied Science,

Committee Member

Man-Lap (Alex) Li, Assistant Professor of Engineering and Applied Science,

Committee Member

©Copyright 2011 by Abdullah Kayi
All rights reserved

Acknowledgments

This dissertation would not have been possible without the guidance, friendship and support of so many people who in one way or another contributed and extended their valuable assistance in the preparation and completion of this study.

First and foremost I offer my sincerest gratitude to my advisor Prof. Tarek El-Ghazawi, who has supported me thoughout my thesis with his patience and knowledge whilst allowing me the freedom to explore in my own way. As the founding director of the High Performance Computing Laboratory (HPCL) at The George Washington University (GWU), co-director of the NSF Industry/University Center for High-Performance Reconfigurable Computing (CHREC) and director of IMPACT (The Institute for Massively Parallel Applications and Computing Technologies), Prof. El-Ghazawi gave me the opportunity to be engaged in a wide range of advanced research projects including the Unified Parallel C (UPC) project and multiple research projects funded by Arctic Region Supercomputing Center (ARSC). Throughout these projects, I have gained unique real-life experience on the state-of-the-art supercomputing systems. In the meantime, his guidance and support helped me mature as a researcher. The scientific skills and experiences acquired during my 6.5 years at HPCL have become an invaluable asset for my future professional career.

I would also like to thank my committee members Prof. Can Korman, Prof. Edward Della Torre, Prof. Alex Li, Prof. Suresh Subramaniam, and Prof. Abdou Youssef for their time and valuable feedback. I am grateful to Prof. Coşkun Bayrak for creating the opportunity for me to start my graduate studies.

I acknowledge ARSC for supporting my research. My summer internships at the Arctic Region Supercomputing Center built the fundamentals of my thesis work. The associated experience broadened my perspective on the practical aspects of my research.

The members of the HPCL group have contributed immensely to my personal and professional time at GWU. The group has been a source of friendships as well as good advice and collaboration. I would like to express my gratitude to all current and past group members: Dr. Vikram Narayana, Dr. Saumil Merchant, Dr. Joseph Schneible, Dr. Esam El-Araby, Dr. Miaoqing Huang, Dr. Proshanta Saha, Dr. Suboh Suboh, Dr. Lubomir Riha, Dr. Mohamed Taher, Dr. Ivan Gonzales, Dr. Mohamed Bakhouya, Dr. Harald Simmler, Ahmed Anber, Teng Li, Lenny Wang, Abdullah Al-Dahlawi, Maria Malik, Mohamed Abou-Ellail, Preetham Nosum. I have enjoyed working with Francois Cantonnet, Yiyi Yao, Kun Xi, Samy Al-Bahra, Ahmed Anber and Olivier Serres in the UPC project. Special thanks to Samy Al-Bahra for being a hard-working team member and co-author in various research papers and software projects that we worked on. Last but not least, I would like to especially thank Olivier Serres for not only being a good friend but also his support and intelligent ideas to solve challenging problems during my dissertation. Our brainstorming discussions were invaluable for my implementations.

Finally, I would like to dedicate this dissertation to my wife Efsun because it would not have happened without her support, encouragement, patience, and love. Along with Efsun, I wish to give my special thanks to my family for all their support during my doctoral education. I would not be here without them.

Abstract

An Efficient Cache Coherence Mechanism for Chip Multiprocessors

Due to power and clocking constraints, integrating more processing cores onto a single chip, instead of increasing the frequency has become the norm in modern processor design. This design choice has resulted in the advent of multicore architectures also referred to as Chip Multiprocessors (CMPs) as the dominant processor architecture.

CMPs introduce many challenges that need to be addressed if we are to harness their potential performance. One such challenge comes with shared-memory processor architectures, which namely is the cache coherence problem. The trend of putting more and more cores inside a single chip makes efficient cache coherence design imperative to get performance scaling out of CMPs. Earlier cache coherence research targeted the multi-processor systems. CMPs have different technological parameters and physical constraints than earlier multi-processor systems, which require a fresh look at the design of cache coherence protocols. Optimizing on-chip resource utilization is crucial for CMPs, as for example a remote but on-chip cache-to-cache access is far less costly than an off-chip access. This is different than earlier multiprocessors where shared resources were always off-chip. On-chip scalable point-to-point interconnects are also emerging for CMPs. With the utilization of such interconnects, relative distance between cores varies a lot in CMPs compared to the traditional multi-processor systems. These topological and technological differences call for a cache coherence protocol design, which maximizes on-chip communication with the purpose of reducing off-chip data transfers.

Contemporary architectures employ write-invalidate based cache coherence protocols that are known to generate coherence misses that later produce additional latency. This is especially true under producer-consumer sharing patterns and can become a performance bottleneck as the number of cores increases. On the other hand, write-update based cache

coherence protocols can solve the coherence misses problem, but they tend to generate excessive network traffic. Previous studies have shown that a single protocol approach is not sufficient for many sharing patterns.

As a solution, this dissertation presents two mechanisms to design efficient and scalable cache coherence protocols for CMPs. First, we propose an adaptive hybrid, invalidate/update, protocol to reduce coherence misses observed in write-invalidate based protocols. The proposed protocol is based on a write-invalidate scheme, however, adaptively it pushes updates to potential consumers based on the observed producer-consumer sharing patterns. Secondly, we extend this adaptive protocol with interconnection resource aware mechanisms. These mechanisms require modest hardware extensions in order to benefit from such adaptive protocols. In addition, the proposed mechanisms can be implemented on top of any write-invalidate protocol and can work with existing language, compiler, and processor designs.

Extensive analytical as well as experimental evaluations, conducted on a tiled-CMP via full-system simulator, were used to assess the performance from our proposed dynamic hybrid protocols. Performance studies used a set of scientific applications from SPLASH-2 and NAS parallel benchmarking suites. The experimental results, confirmed by the analytical model, have shown that the proposed mechanisms reduce cache-to-cache sharing misses up to 48% and in return speed up application performance up to 25%. In addition, the proposed interconnection-resource-aware mechanism is proven to perform well under varying interconnection loads.

Table of Contents

Acknowledgments **iv**

Abstract **vi**

Table of Contents **viii**

List of Figures **xiii**

List of Tables **xvii**

List of Acronyms **xix**

1 Introduction **1**

1.1 Background Information 2

1.1.1 Emergence of Chip Multiprocessors 2

1.1.2 Cache Coherence 4

1.1.2.1 Snooping Protocols 5

1.1.2.2 Directory-Based Protocols 6

1.1.2.3 Cache Coherence States 8

1.2 Motivations 11

1.3 Problem Statement 14

1.4 Scope and Limitations 15

1.5 Proposed Approach 15

1.6 Organization of Dissertation 18

2 Related Work **19**

2.1 Cache Coherence Techniques 19

2.2 Adaptive/Hybrid Cache Coherence Protocols 23

2.3 Cache Coherence Predictors 25

2.4 Producer-Consumer Communication 26

2.5 Cache Coherence Optimizations for Chip Multiprocessors 27

3 Evaluation Strategies and Testbeds **29**

3.1 Evaluation Methodology 30

3.2 Experimental Workloads 33
3.2.1 Benchmarking Suites 33
3.2.1.1 HPC Challenge Benchmarks 33
3.2.1.2 LMbench 34
3.2.2 Applications 35
3.2.3 Simulation Workloads 36
3.3 Simulation Strategies 37
3.4 Testbeds 38
3.4.1 Intel Core 2 Duo E6600 (Conroe) 40
3.4.2 Intel Core 2 Quad Q6600 (Kentsfield) 41
3.4.3 Intel Xeon E5310 (Clovertown) 42
3.4.4 AMD Opteron 2218 (Santa Rosa) 42
3.4.5 AMD Opteron 2356 (Barcelona) 42
3.4.6 Sun UltraSPARC T1 (Niagara) 43
3.4.7 Sun UltraSPARC T2+ (Victoria Falls) 44
3.4.8 Sun Fire Cluster (Midnight) 44
3.5 Analytical Models 47

4 Cache Coherence Impacts in Modern Architectures: An Empirical Study 48

4.1 Experimental Results From Emerging CMPs 49

4.1.1 Memory Bandwidth Analysis 50

4.1.2 Memory Latency Analysis 58

4.1.3 Cache Coherence Overhead Analysis 59

4.2 Analyzing the Performance Impact of Cache Coherence in CMP-based systems 61

4.2.1 AMD Cache Coherence Protocol 62

4.2.2 Experimental Setup Overview 63

4.2.3 Performance Scaling Analysis 65

4.2.4 Memory Bandwidth Analysis 66

4.2.5 Memory Latency Analysis 69

4.2.6 Operating System (OS) Related Analysis 70

4.2.7 Performance Analysis Using Hardware Counters 72

4.3 Summary 74

5 Producer-Consumer Adaptive Cache Coherence Protocols for CMPs 76

5.1 Design Considerations 76

5.1.1 Tiled-CMP Architecture 79

5.1.2 Baseline Cache Coherence Protocol 82

5.2 Key Concepts and Mechanisms 82

5.2.1 Producer-Consumer Predictor Cache 83

5.2.2 Producer-Consumer Sharing Pattern Detection 84

5.2.3 Bandwidth Adaptive Mechanism 86

5.2.4 Proximity Aware Mechanism 88

5.3 Protocol Implementation Details 89

5.3.1 Trigger Update Mechanism 93

5.4 Experimental Verification of the Protocols 94

5.5 Analytical Model 95

5.5.1 Assumptions 97

5.5.2 The CMP system model 98

5.5.3 Data Access Patterns and Parameters 99

5.5.4 Cost Calculation of Data Access Patterns 99

6 Performance Results for the Producer-Consumer Adaptive Cache Coherence Protocols 102

6.1 Experimental Plan 102

6.1.1 Simulation Environment 103

6.1.2 Simulation Workloads 104

6.2 Cache Coherence Protocol Performance Results 105

6.2.1 Bandwidth Adaptivity Analysis 107

6.2.2 Proximity Aware Protocol Analysis 110

6.2.3 Sensitivity Analysis 114

6.2.3.1 Trigger Update Delay Latency Sensitivity Analysis . . . 114

6.2.3.2 Analytical Model Results 115

7 Summary and Conclusions **117**

7.1 Summary 117

7.2 Summary of Contributions 119

7.3 Future Work 123

List of Figures

1.1 A sample multiprocessor system 3

1.2 Permissible MOESI States 8

1.3 Characterization of the MOESI States 8

1.4 MOESI State Transition Diagram with a Snooping Protocol 9

2.1 Classification of cache coherence protocols based on the coherence enforcing mechanism 20

2.2 Classification of cache coherence protocols based on write-policy to shared data 23

3.1 High-level classification of the experimented CMP systems 30

3.2 Experimented CMP systems based on the Instruction-Set Architecture (ISA) 30

3.3 Architectural overview of some of the experimental testbeds 39

3.4 Architectural overview of thin and thick nodes in Midnight cluster 45

4.1 STREAM TRIAD Results 51

4.2 Comparison of peak STREAM Results 52

4.3 Memory Copy Bandwidth Results for Different Data Sizes 53

4.4 Memory Read Bandwidth Results for Different Data Sizes 54

4.5 Memory Read-Write Bandwidth Results for Different Data Sizes 55

4.6 Memory Write Bandwidth Results for Different Data Sizes 56

4.7 Memory Read Latency Results 59

4.8 Normalized Cache Coherence Misses 60

4.9 Application performance comparison between thin nodes and thick nodes . 63

4.10 FFT Results (thin nodes vs. thick nodes) 65

4.11 Effective Bandwidth Results (thin nodes vs. thick nodes) 66

4.12 STREAM TRIAD results by cpu binding (array size = 1,5M) 67

4.13 Memory Read Latency results from a thick node 69

4.14 Application Performance Comparison Junctions vs. ChaNGa 71

4.15 L2 Cache Statistics of Junctions and ChaNGa 72

4.16 STREAM TRIAD results by cpu binding (Array Size = 20K) 73

5.1 Write-Invalidate vs. Write-Update protocol 78

5.2 TILE64 Architecture Diagram 80

5.3 Target tiled-CMP architecture 81

5.4 Extended Directory-Cache/PCPC Fields 84

5.5 Adaptive Cache Coherence mechanism flowchart 86

5.6 Target tiled-CMP architecture for PCPAM protocol study 88

5.7 Write-Invalidate Directory protocol 90

5.8 Related Work 91

5.9 Producer-Consumer Adaptive Multicore (PCAM) protocol 92

6.1 Ratio of potential consumers out of all LOADs 105

6.2 Cache coherence misses improvement 106

6.3 Execution time improvement 107

6.4 Bandwith adaptivity results 108

6.4 Bandwith adaptivity results 109

6.5 Performance Results from a 64-core tiled-CMP 111

6.6 Update Trigger delay sensitivity analysis 112

6.6 Update Trigger delay sensitivity analysis 113

6.7 Performance Results with Analytical Model 114

6.8 Performance Results from 256-core tiled-CMP 115

List of Tables

1.1 The Cache Coherence Problem 5

3.1 CMP specific performance analysis metrics 31

3.2 Simulation Workloads 36

3.3 Simulation Parameters 37

3.4 System Characteristics of the Experimental Testbed 38

3.5 System characteristics of the experimental testbed - Sun Fire cluster 46

4.1 Numastat Output 71

5.1 Simulation Architectural Details 81

5.2 Data Access Patterns and Parameters 96

5.3 System Events 96

5.4 Probabilities and Corresponding Costs for System Events for the MRMW Pattern 97

6.1 Simulation Workloads 104

List of Acronyms

ACK	Acknowledgement
AMD	Advanced Micro Devices
AMR	Adaptive Mesh Refinement
API	Application Programming Interface
ARSC	Arctic Region Supercomputing Center
AVUS	Air Vehicles Unstructured Solver
BASH	Bandwidth Adaptive Snooping Hybrid
ccNUMA	cache-coherent Non Uniform Memory Access
ChaNGa	Charm N-body Gravity solver
CMP	Chip Multiprocessor
CMT	Chip Multi-threading
CTH	Computational Hydrodynamics and Radiative Thermal Diffusion squared to the Three-Halves
DC	Directory Cache
DoD	Department of Defense
DRAM	Dynamic Random-Access Memory
DSM	Distributed Shared Memory
FFT	Fast Fourier Transform
FLOPS	FLoating-point OPerations per Second

FSB	Front Side Bus
FPU	Floating Point Unit
HPC	High-Performance Computing
HPCC	High-Performance Computing Challenge
HPCMP	High-Performance Computing Modernization Program
HT	HyperTransport
HW	Hardware
HYCOM	Hybrid Coordinate Ocean Model
IBM	International Business Machines
ICEPIC	Improved Concurrent Electromagnetic Particle In Cell
ILP	Instruction Level Parallelism
I/O	Input Output
ISA	Instruction Set Architecture
ITRS	International Technology Roadmap for Semiconductors
GEMS	General Execution-driven Multiprocessor Simulator
GETS	Get Shared
GETX	Get Exclusive
L1	Level 1
L2	Level 2
L3	Level 3
LAMMPS	Large-scale AtomicMolecular Massively Parallel Simulator
LRU	Least Recently Used
MCM	Multi-Chip Module
MESI	Modified, Exclusive, Shared, Invalid
MOSI	Modified, Owned, Shared, Invalid
MOESI	Modified, Owned, Exclusive, Shared, Invalid
MP	Multiprocessor

MR	Multiple Reader
MRMW	Multiple Reader Multiple Writer
MRSW	Multiple Reader Single Writer
MT	Multi-threading
MW	Multiple Writer
NAMD	NAnoscale Molecular Dynamics
NPB	NAS Parallel Benchmarks
NUMA	Non Uniform Memory Access
OS	Operating System
PAPI	Performance Application Programming Interface
PCAM	Producer-Consumer Adaptive for Multicores
PCBAM	Producer-Consumer Bandwidth Adaptive for Multicores
PCPAM	Producer-Consumer Proximity Adaptive for Multicores
PCPC	Producer-Consumer Predictor Cache
PGAS	Partitioned Global Address Space
PUTX	PUT Exclusive
SIMD	Single Instruction Multiple Data
SMT	Simultaneous Multi-threading
SR	Single Reader
SRMW	Single Reader Multiple Writer
SRSW	Single Reader Single Writer
SSE	Streaming SIMD Extension
SPLASH	Stanford Parallel Applications for Shared-Memory
SMP	Symmetric Multiprocessor
SW	Single Writer
TLP	Thread Level Parallelism
UMA	Uniform Memory Access

WB	Write-Back
WRF	Weather Research and Forecasting

Chapter 1

Introduction

Chip Multiprocessors (CMPs) have become the norm in processor design. This transition provides new ways of reaching higher performance without increasing the power requirements [1, 2, 3, 4]. Most CMPs adapt the shared memory model to achieve a simpler programming environment. As such, cache coherence becomes a crucial design issue and performance scaling becomes a must. A background on the emergence of CMPs will be first provided here, followed by an overview of the cache coherence problem and various implementation options. The motivations for the research work conducted in this thesis will be discussed. This will be followed by a formal statement of the research problem, an overview of the proposed methodology, as well as the scope and limitations of this work. The organization of this thesis will be provided at the end of the chapter in order to provide an overview of the detailed research work that was carried out and its findings.

1.1 Background Information

1.1.1 Emergence of Chip Multiprocessors

The demand for computing power continues to increase in virtually every domain, from the basic desktop systems to the high-end computing platforms. Advancement in processor design is largely driven by Moore's Law, which predicts that the number of transistors per integrated circuit area roughly doubles every two years [5]. While Moore's Law is expected to continue at least into the next decade, using all these transistors for a single processing core is facing practical challenges. These challenges include power dissipation and limited instruction-level parallelism [1, 6].

Historically, performance increase in processors was mainly reached by increasing the clock frequency at which the processors run and by increasing the amount of parallelism [7, 8]. As transistor density per silicon area increases, so does the ability to increase device complexity and integrate more features onto a single chip. This resulted in different ways of increasing parallelism. Early advancement of microprocessors include the widening of the data path which has increased currently to 64-bits. Another enhancement in parallelism has been through executing multiple instructions at the same time (instruction-level parallelism or ILP) via pipelining techniques and superscalar architectures.

Unfortunately, it is becoming increasingly difficult for processor designers to exploit these techniques for better performance. Due to the limited instruction-level parallelism, superscalar processors do not achieve much benefit by issuing more than four instructions per cycle [8]. In addition, design complexity and power requirements are major issues that limit further performance gains from ILP [9]. Accordingly, major microprocessor vendors have shifted their designs into utilizing the available number of transistors inside a single

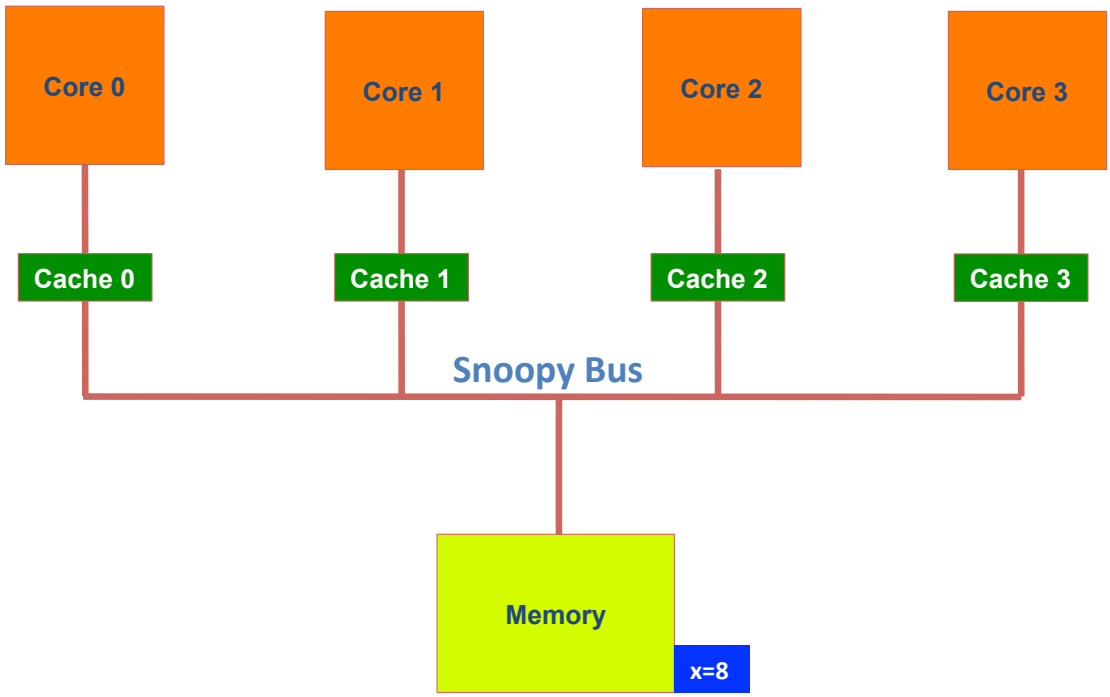


Figure 1.1: A sample multiprocessor system

chip by using multiple processors (cores) resulting in *Multicore* architectures, also known as *Chip Multiprocessors* (CMPs). These architectures provide a solution to increase the performance capability on a single chip without requiring a complex architecture which increases power requirements [1, 2, 3, 4]. Therefore, CMPs have become the dominant processor architecture for both desktop and high-performance platforms. Several chip manufacturers including AMD, IBM, Intel, and Sun have released systems, that have multiple processing cores on a single chip. This trend will likely continue with more and more cores being placed on a single chip [4, 10, 11]. Thereby, not only high-end computing systems but also the commodity computing platforms will include a certain level of shared memory system with multiple processing cores.

1.1.2 Cache Coherence

Multiple levels of caches have been used to improve performance by hiding memory access latencies and by reducing the bandwidth demands of a processor. In a multiprocessor or similarly multicore system, this local caching of data results in the *cache coherence problem*. This happens because the memory view from different processors is through their individual caches, which might change in time, as any processor can update the values in their local caches [12]. Table 1.1 shows an example of a cache coherence problem that can arise for the system demonstrated in Figure 1.1. This example is inspired from a similar example given in [12] with write-through caches. Initially, X is only stored in the memory with value 8. At time step 1, Core 1 reads X; then at time step 2, Core 2 reads X. The cache coherence problem happens after Core 1 stores a new value 5 into X at time step 3. Without a preventive mechanism at time step 4, Core 2 will end up getting a stale copy of X. This problem is referred to as the *cache coherence problem*. To resolve such a problem, a mechanism called *cache coherence protocol* is utilized to keep all shared copies up to date and to ensure a read request will always get the most recent value. The cache coherence protocol establishes the coherence by responding a with proper action whenever a cache line L is written [13]. There are two main classes of cache coherence protocols according to how they respond when a cache line is written: *write-invalidate protocol* or *write-update protocol*. In an invalidate protocol, the cache coherence problem presented in Table 1.1 would be resolved by invalidating the line containing X in Core 2's cache. As such, at time step 4, Core 2 actually suffers a cache miss instead of reading a stale copy. On the other hand, an update protocol sends the newly written lines to all the sharers so that each core keeps updated data in its local cache. In this case, at time step 4, Core 2 would directly read the value of X from its own local cache. In either scenario, the cache coherence problem would be eliminated and Core 2 gets the most recent value of X.

Table 1.1: The cache coherence problem for a memory location(X) in a multiprocessor system as depicted in Figure 1.1.

Time	Event	Cache 1	Cache 2	Memory
0				8
1	Core 1 reads X	8		8
2	Core 2 reads X	8	8	8
3	Core 1 writes 5 into X	5	8	5
4	Core 2 reads X	5	8!	5

The cache coherence protocol can be implemented in *hardware* or *software*. Most modern architectures use a hardware based solution, and accordingly in this thesis our implementations are all based on a hardware cache coherence solution. As illustrated in the earlier example, a cache coherence protocol must locate all cached copies of a memory location and accordingly should eliminate stale copies [12]. To do so, there are two major ways of enforcing coherence: *broadcast-based protocols* (*snooping protocols*) or *directory-based protocols*.

1.1.2.1 Snooping Protocols

Snooping protocols, also known as broadcast-based protocols, are the most common way of enforcing coherence in small scale multiprocessor and multicore architectures with a single shared memory. This is mainly due to the fact that these protocols can use the same connection, typically a shared bus, to establish a coherent view [12]. Having a bus like ordered interconnect makes the coherence implementation easier. Every cache that shares a data block from memory also keeps the sharing status of the same block. In such a protocol, a broadcast message is sent directly to all processors to locate the owner of a block [14]. The current owner is determined with the snooping mechanism as all cache controllers snoop on the broadcast medium (a bus or switch) [12] to determine whether they are responsible for the block that is requested. Accordingly, a direct cache-to-cache

transfer can be achieved for the requested data block which is an advantage over directory protocols. However, snooping protocols rely on ordered interconnects to maintain total ordering of coherence transactions which do not scale well [15]. In addition, broadcast messages generated for each cache miss increase the bandwidth requirements. Therefore, large scale systems deploy directory-based protocols.

1.1.2.2 Directory-Based Protocols

Directory-based protocols provide a scalable alternative to broadcast-based snooping protocols. The sharing status of each cache line is stored in a location called the *directory*. As such, the directory holds the information for each cache line including the sharers of the specific cache line if any, or for an exclusive line which cache has the latest copy [13]. For each cache miss, the directory is utilized to maintain coherence by taking the proper response according to the request type and the current state of the directory [13].

In a directory-based protocol, the following main classification is utilized to differentiate processors [16]:

- The *home node* is the processor which holds the requested block in its memory.
- The *exclusive node* is the processor which holds the most recent copy of the cache block.
- The *requesting node* is the processor that requests the cache line after a miss in its own cache.

Whenever a processor wants to read a cache line L, it must first send a request to the home node of L. The home node will then respond with an appropriate action. For example, if the home node has a clean copy of the line L, the requesting processor will be added to the

sharers list and the home node will forward a copy of the line *L* to the requester. If some other processor (the exclusive node) has modified the cache line *L* and got the exclusive copy of the line *L*, then the home node forwards the request to that processor to supply the requested data block [13].

Directory-based protocols do not rely on an ordered interconnect to establish total ordering of transactions, as this is achieved via the directory itself and explicit message acknowledgments [15]. Accordingly, highly scalable interconnects can be used with a directory protocol. As discussed earlier, directory is utilized as an indirection point which obviates the need for broadcast messages. This mitigates the bandwidth requirements but creates extra latency for cache-to-cache misses especially when the home node is not the actual owner of the cache line. With the overhead of the directory structure, directory-based protocols have slightly larger implementation costs and hardware requirements but in return, and most importantly, they can scale to larger processor counts [12]. Considering the increasing number of cores in CMPs, scalability is an important issue, and thereby we use directory-based protocols to enforce coherence in this thesis. In order to prevent the directory from becoming the bottleneck, we also utilized a distributed directory scheme as in the case with Distributed Shared Memory (DSM) machines. The main disadvantage of such a directory-based write-invalidate protocol is the 3-hop misses required to complete the events that are generated after L2 cache misses by requesting nodes where the home node does not have the latest copy. Our proposed protocols target this bottleneck by alleviating the latency burden via data-forwarding protocol extensions to establish write-updates for lines that are deemed to be exhibiting a producer-consumer sharing pattern.

	M	O	E	S	I
M	✖	✖	✖	✖	✔
O	✖	✖	✖	✔	✔
E	✖	✖	✖	✖	✔
S	✖	✔	✖	✔	✔
I	✔	✔	✔	✔	✔

Figure 1.2: Permissible MOESI states at a given time between two processors.

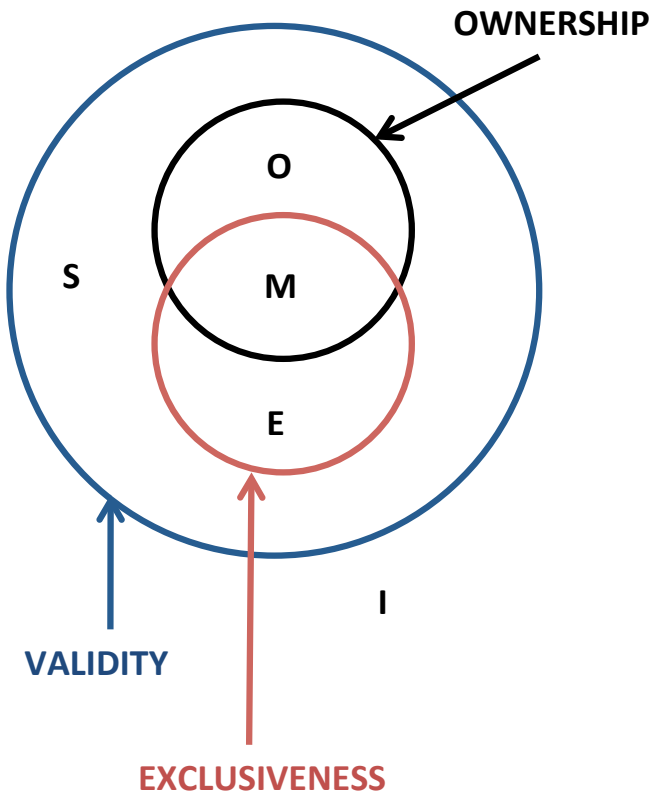


Figure 1.3: Characterization of the MOESI States [17].

1.1.2.3 Cache Coherence States

Cache coherence protocols use protocol states to differentiate various read/write permission scenarios for the shared data blocks [19]. They are also often used to classify protocols, such as the two most common cases, the MESI protocol and the MOESI protocol.

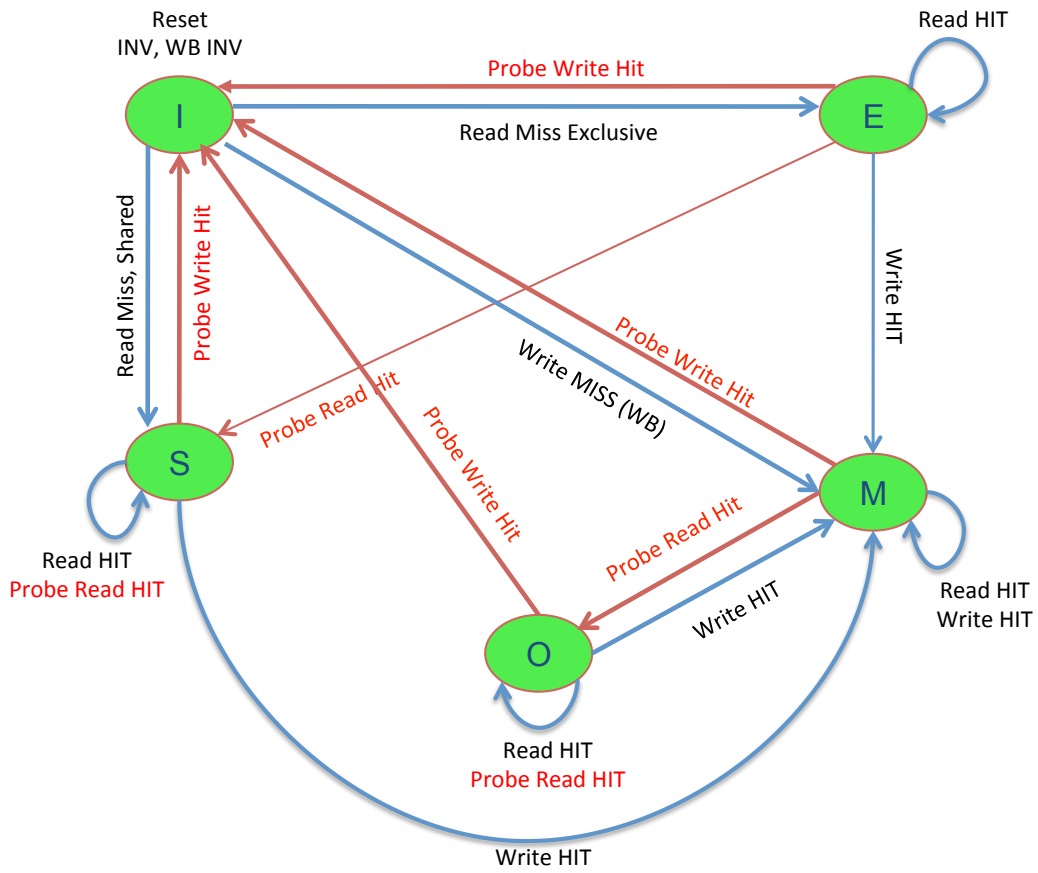


Figure 1.4: MOESI State Transition Diagram with a Snooping Protocol [18].

Both of these protocols are named after the main states they represent (Modified, Owned, Exclusive, Shared, Invalid). The difference between these two protocols comes from the O(Owned) state which was introduced by Sweazey and Smith [17]. Since MESI protocol states are a subset of the MOESI protocol, in this section we only focus on the MOESI protocol discussion which in most part is valid also for the MESI protocol.

The states of the MOESI protocol as described in AMD64 documentation [18] are as follows:

- *Invalid*: Either the data block is not present in the cache or the cache line does not have the valid entry. Valid copies of the data can be either in main memory or at another processor cache.

- *Exclusive*: This state shows that the corresponding cache line holds the most recent, correct copy of the data. Also, the copy in the main memory is valid. Only a single processor can be at this state at a given time.

- *Shared*: This state is similar with the exclusive state except other processors may also have the same copy. Also, to have the recent copy in the memory no other processor should hold the line in the *owned* state.

- *Owned*: Similar to the *shared* state but the copy in main memory can be invalid. At any given time, only one cache line can be in the owned state, the other processors must be in the shared state at that time. The *owner* cache is the one responsible for future write-backs to memory since the memory might have a stale copy.

- *Modified*: The cache line in this state holds the most recent copy and no other processor has a copy.

Figure 1.2 shows the permissible states at a given time for a cache line among different caches by demonstrating a case for a pair of processors for simplicity. For instance, if a cache line is in the *exclusive* state, from the figure it is shown that the other lines can only be in *invalid* state.

Sweazey and Smith [17] made a classification of these five states based on the cache line's validity, exclusiveness, and ownership. Corresponding characterization is demonstrated in Figure 1.3. Modified, Owned, Exclusive, and Shared states have cache lines that are considered *valid*. Cache lines that are in Modified and Exclusive states have *exclusive* data, that is contained in one and only one cache [17]. *Ownership* of a data block means as being responsible for its accuracy/validity [17]. Owner of a cache line is responsible for updating the memory and responding to data requests for that line. Figure 1.4 illustrates the MOESI state transition diagram for a snooping protocol implemented in AMD64 platforms [18]. Although some state transitions are implementation specific, this diagram is generic enough

to represent the relation between the MOESI states in terms of various read and write actions.

1.2 Motivations

Years of continued increase in single-thread performance mainly through the increase in processor clock frequency has ended. This is described as "The Free Lunch is Over" which shows the fundamental turn towards concurrency to achieve performance [20]. The start of this *multicore era* is providing new ways to reach higher performance via thread-level parallelism (TLP) and accordingly parallel programming. This programming paradigm shift will impose a burden on software programmers. Shared-memory programming models or similarly Partitioned Global Address Space (PGAS) programming models provide relatively easy parallel programming solutions to alleviate this burden. As such, most of the parallel software that runs on CMPs is based on some flavor of the shared-memory programming model. Although processors logically access the same globally addressable shared memory, on-chip cache hierarchies should be exploited to achieve better performance on CMPs. Having various cache hierarchies in CMPs brings the well-known cache coherence problem which was discussed earlier in Section 1.1.2. To resolve this problem entails developing a mechanism to provide a consistent view of the memory with various cache hierarchies, which is handled via a cache coherence protocol. This *cache coherence protocol* is a crucial correctness and performance-sensitive design point for supporting the shared-memory model [21]. As more and more cores are put on a single-chip, the cache coherence problem will become a key performance bottleneck in CMPs as it was experienced in earlier SMP/ccNUMA systems. We have shown the severe effects of cache coherence overhead on application performance including large high-performance clusters with multi-socket CMP based shared memory nodes [22]. More details from our study on

the performance impact of cache coherence overhead in modern architectures can be found in Chapter 4. The conclusion was that, it is imperative to continue innovation in cache coherence mechanisms to gain better performance from CMPs.

Cache coherence has been investigated heavily in the research community, but most of the prior work focused on multiprocessor systems (MPs) comprised of multiple single-core processors [21]. With the advent of CMPs, the cache coherence problem became a first-order design issue at the chip level. Many constraints including interconnect, latency, power, and energy differ in MP and CMP designs. The main differences can be summarized as follows:

- Latencies between cores in CMPs are far less compared to typical SMPs.
- The relative latency difference between nodes was not significant in typical MPs but it is now in CMPs.
- From any core/node in the system, memory is no longer at a comparable distance as the neighboring cores/nodes. Off-chip memory access latency is much higher than on-chip cache-to-cache access latency.
- The inter-core bandwidth on a CMP can be much higher compared to a traditional SMP.

According to the International Technology Roadmap for Semiconductors (ITRS), the off-chip bandwidth is expected to increase linearly rather than exponentially. Thus, a high on-chip cache performance is imperative to cut down on bandwidth [23]. In addition, interconnection issues are different in CMPs as compared to traditional MPs [24, 25]. On a traditional ccNUMA multiprocessor, the latency from a given node to the closest neighboring node and to the farthest node are often within a factor of two of each other [24]. This

is mainly because the latency is dominated by off-chip and sometimes off-board latencies. However on a CMP, the relative latencies vary significantly albeit the smaller absolute latencies. As such, to communicate with a core that is five hops away takes much longer than with a one that is a single hop away. Having said that, the latency between cores is still small in absolute terms on CMPs, and the dominant latency bottleneck is to off-chip memory. Furthermore, considering the trend of putting more cores inside a single chip, cache-to-cache latency mainly caused by coherence misses will become a crucial performance bottleneck in future CMPs. Thus, solutions to reduce remote miss latencies have become important for CMP performance. All these differences make it important to revisit the design of scalable cache coherence protocols for the CMPs in this new context [23].

Cache coherence schemes in general are tightly coupled to the ordering properties provided by the underlying interconnect [21]. Buses provide simpler interface with their total ordering properties but will suffer to scale after a moderate number of cores. As the core numbers increase inside a single CMP; low-power, high-bandwidth, scalable on-chip communication substrate is imperative to enable scaling to large core counts [26]. Furthermore, on-chip interconnects for future CMPs will have different design constraints with tighter resource budget compared to the earlier earlier multiprocessors [21]. Therefore, not only the cache coherence protocols for future CMPs should be able to work on unordered interconnects but also adaptive mechanisms will be critical to adjust varying interconnection resources.

Most of the contemporary architectures utilize write-invalidate cache coherence protocols. The main reasons for this and more discussion including the write update-protocols, will be given in Section 1.5. Unfortunately, write-invalidate protocols tend to generate high coherence misses or remote cache-to-cache misses. As mentioned earlier in contrast to traditional multiprocessor systems, relative latency between cores will vary a lot as the number of cores increases within a CMP. Moreover, it may require dozens of cycles for

a signal to traverse between two opposite edges of the die [21, 23]. Thereby, solutions to alleviate the burden of these remote misses under write-invalidate protocols will become important in future CMPs.

1.3 Problem Statement

This research is targeting the cache coherence problem in CMP based systems. While the shared-memory view adapted by CMPs provides a relatively easy programming model compared to distributed memory systems, it brings up the cache coherence problem. With the increasing number of cores inside a single chip and utilization of multiple CMPs, the cache coherence problem becomes more crucial for better performance. Therefore, given a CMP based system with shared resources it is required to:

- Empirically analyze and characterize the performance impact of cache coherence overhead
- Introduce efficient solution(s) to mitigate the effects of cache coherence overhead
- Provide efficient mechanisms to integrate these solutions to the underlying hardware
- Optimize the proposed solutions by leveraging the underlying interconnect characteristics
- Conduct extensive analytical and simulation studies to assess the utility of the proposed solutions

1.4 Scope and Limitations

Our preliminary work has revealed the major performance issues in CMP based systems, and characterized the performance impact of maintaining the cache coherence among multiple caches in such platforms. Accordingly, this research is mainly targeting the remote miss latencies induced by the current cache coherence protocols by focusing on reducing the coherence misses. In specific, our proposed adaptive cache coherence protocols are based on dynamic detection of producer-consumer sharing pattern inside the running workload. This allows speculative updates that reduce coherence misses. As such, our proposed protocols allow on-chip updates to reduce miss penalties. Our prediction mechanism requires a directory like structure as it currently utilizes the directory controllers inside the chip. We also assume that some level of the cache hierarchy inside CMP architectures includes private caches.

1.5 Proposed Approach

Experimental Methodology: The main goal of this research is to analyze and optimize CMP based architectures from cache coherence perspective. To achieve this goal and the objectives stated in Section 1.3, we have started with a comprehensive performance analysis study of CMP based systems to isolate performance bottlenecks. We have also characterized the performance impact of cache coherence overhead on system performance. Based on the observations from the preliminary experimental work on performance analysis, we have formally modeled the architectural components pertaining to major performance bottlenecks using a well-known simulator infrastructure with focus on cache coherence. This research is conducted with full-system simulation utilizing Virtutech Simics [27] extended with the GEMS toolset [28]. A full-system simulator was chosen to include operating sys-

tem effects in order to establish a more practical platform for our studies. Simics provides the full-system functional simulation and GEMS provides a detailed memory system timing infrastructure which observes all cache coherence protocol messages and corresponding state transitions.

Protocol Implementation: Broadcast-based snooping protocols and directory-based protocols are two main choices to implement cache coherence protocols. Although snooping protocols provide lower latencies as targeted in this paper, they also generate heavy traffic and put too much burden on the interconnect. Furthermore, they usually require an ordered network such as bus interconnect which can be another performance bottleneck. As such, bus-based systems are not scalable. Hence, in this work we implemented our protocols using a directory scheme. Also, our study looks at the cache coherence problem from the hardware perspective and, thus, implements the proposed mechanisms as a hardware-based cache coherence protocol.

Cache Coherence Optimizations: Based on the policy during a write operation on shared data, cache coherence protocols fall into two main categories, *write-update* based protocols and *write-invalidate* based protocols. Write-update protocols involve interprocessor communication on every write operation to shared data. Write-invalidate protocols maintain cache coherence by invalidating copies of a memory block when the block is modified by a processor. The advantage of this is that at most the first write, in a sequence of writes to the same block with no intervening read operations from other processors, causes interprocessor communication. Hence, subsequent write operations can be completed locally until the same data block is requested by another processor. As such, write-invalidate protocols yield less network traffic and accordingly provide a better scalable solution to the cache coherence problem but they suffer from the high read latency induced by cache co-

herence misses. On the other hand, previous research has shown that coherence misses or cache-to-cache misses comprise a big portion of overall cache misses in shared memory systems [16, 29, 30, 31, 32] and write-update protocols can reduce these coherence misses [33, 34, 35]. However, write-update protocols tend to generate high traffic on the interconnect due to unnecessary updates especially for migratory objects. So, a single cache coherence policy solution to effectively provide performance for different types of applications is not feasible. Having said that, it is difficult to implement a write-update based protocol on unordered interconnects and in general performance improvements from the protocol are offset by having a sequentially consistent memory system for these protocols. Therefore, most contemporary systems implement a write-invalidate based cache coherence protocol.

As a solution to alleviate the coherence misses' impact on performance, we propose to extend the existing directory controllers inside the chip to dynamically detect data sharing patterns of running workloads. Current optimizations are targeting the detection of producer-consumer sharing patterns. As such, we developed a prediction mechanism to identify potential consumers for a cache line. In addition, this prediction mechanism needs to be tightly coupled with the cache coherence protocol to trigger speculative updates to these consumers. Thus, we propose to extend an existing write-invalidate protocol that can support these write-update optimizations. The resulting protocol works as an adaptive cache coherence protocol based on the data sharing pattern of the application dynamically by utilizing the feedback from the prediction mechanism. The goal is to have a simple approach with modest hardware requirements to avoid large design space modifications. As such, the proposed mechanisms do not change the memory model seen by the programmer and compiler. The proposed adaptive protocols can be implemented on top of any write-invalidate protocol and can work with existing language, compiler, and processor designs.

Resource aware optimizations: Furthermore, the designed adaptive cache coherence protocol is enhanced with bandwidth and latency adaptations, based on the characteristics of the underlying interconnection network.

Analytical performance model: Finally, we adapted a high-level performance model to assess the potential gains from such dynamic adaptations and provide a fast tool for design space explorations.

1.6 Organization of Dissertation

This dissertation is organized as follows. Chapter 2 provides a discussion of related work in the context of cache coherence techniques, adaptive/hybrid cache coherence protocols, cache coherence predictors, producer-consumer communication and cache coherence optimizations for CMPs. Chapter 3 describes our experimental work in characterizing the performance issues in CMPs focusing mainly on the cache coherence overhead. The experimental results from this study are given in Chapter 4 covering various modern architectures based on the CMP technology. Also, a simulation study to assess cache coherence overhead is presented in Chapter 4. Chapter 5 introduces our adaptive, dynamic, hybrid cache coherence protocols, and our adapted analytical model. Then, Chapter 6 provides the experimental methodology, simulation workloads, and the corresponding experimental results and analyses. Finally, Chapter 7 provides a summary and concluding remarks.

Chapter 2

Related Work

Computer systems will increasingly use *Chip Multiprocessors* (CMPs) and provide globally addressable shared memory. Such systems will have caches that must be kept coherent. Cache coherence is an important design point for performance and correctness in order to support the shared memory model [21]. This chapter discusses related works for this thesis in the area of cache coherence. The related work includes cache coherence techniques, adaptive/hybrid cache coherence protocols, cache coherence predictors, producer-consumer communication and cache coherence optimizations for CMPs.

2.1 Cache Coherence Techniques

There are two major classes of cache coherence protocols in terms of their implementation, broadcast-based snooping protocols and directory-based protocols. In addition, token-based protocols are also studied mainly in academia [32, 36, 37, 38]. Figure 2.1 shows a high-level classification of cache coherence protocols based on the underlying mechanism that enforces cache coherence.

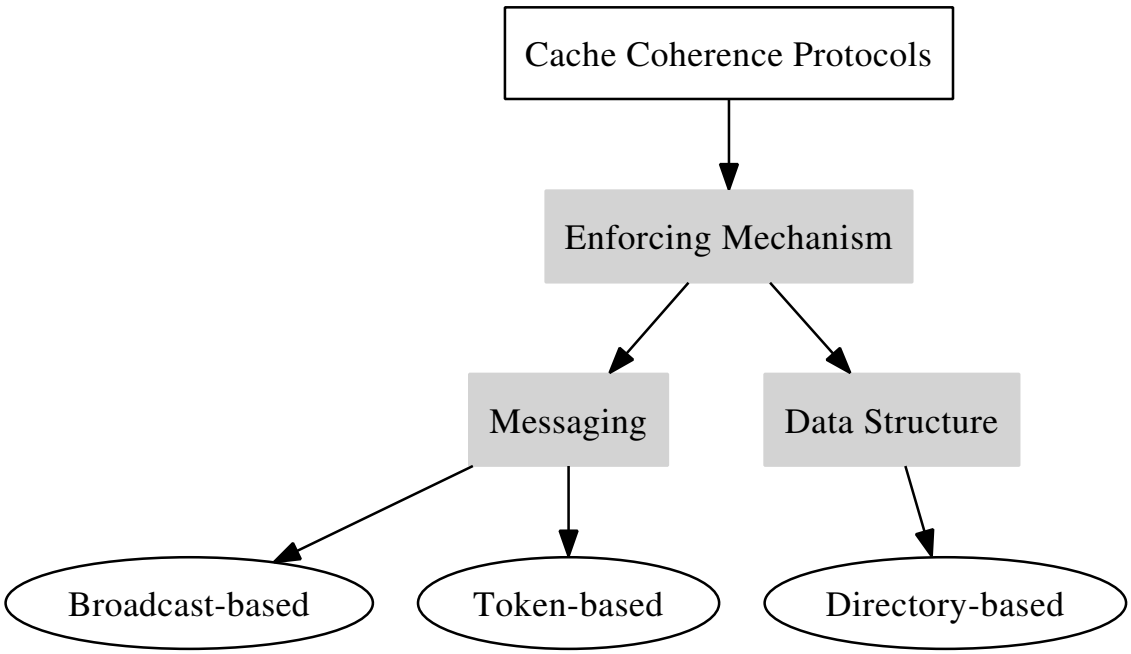


Figure 2.1: Classification of cache coherence protocols based on the coherence enforcing mechanism

In general, those protocols can use a data structure (Directory) to check on the status of a cache line, or use messaging (e.g. token or broadcast based) to communicate such a status at a time of potential change. Broadcast-based snooping protocols have been the dominant cache coherence technique for small scale shared-memory architectures. In such systems, mostly a shared bus connects all processors and memory components [19]. In these protocols, the bus provides key ordering and atomicity properties that eventually simplifies the coherence protocol implementation [21]. However, such ordered interconnects do not scale well with the increasing number of processors. In addition, snooping traffic is based on broadcast messages that require high bandwidth. This limits the scalability of snooping protocols. To mitigate this bandwidth overhead, split transaction snooping buses were proposed [39, 40]. For example, the Sun Starfire system includes four interleaved snoop buses that can work in parallel [40]. In such a system, usually the physical memory is divided into parts and each bus snoops on a specific part of the memory. Also, the Starfire system maps the data and address traffic onto distinct wires to prevent the addresses and data from competing with each other for transmission [40]. Many studies have targeted snoop filter-

ing to alleviate the burden of broadcast messages on the network [41,42,43,44,45]. Snoop filtering is usually done via a hardware mechanism that filters out incoming snoop requests for addresses known not to be in local cache [46]. Commercial products like IBM's Blue Gene supercomputer, e.g. utilizes such snoop filtering techniques to improve its snooping protocol performance [46].

Point-to-point links with a switch based interconnect is a viable alternative to shared buses, and can scale to larger processor counts with good performance [47]. Accordingly, on-chip interconnection networks were suggested to replace on-chip buses [48]. One simple option to form such a point-to-point interconnection is to use a unidirectional ring topology [19,21]. In a ring topology, each node is connected to two other nodes forming a closed loop. Although coherence messages are serialized along the ring, the order of transactions may depend on the node's position on the ring which is different from the bus-based systems [21]. Barroso and Dubois [49,50] studied the performance of a snooping protocol on a logically-embedded slotted ring and proposed a coherence solution for such interconnects. Rings were also used in commercial products like IBM Power4 [51], IBM Power5 [52] and IBM/Sony/Toshiba Cell systems [53]. Later, Marty *et al.* proposed a new ring protocol called *RING-ORDER* that completes requests in the order of ring position without the overhead of an ordering point and without retries [54]. In addition, flexible snooping algorithms were proposed by Strauss *et al.* [55] that adaptively select the best snooping algorithm for a snoop request in a ring-based system.

The trade-off between broadcast-based snooping protocols and directory-based protocols were examined with respect to bandwidth and latency [14, 56, 57]. Bilir *et al.* [56] proposed a new coherence method, *Multicast Snooping*, to convert broadcast messages in a snooping protocol to multicast messages to selected processors. Multicast "mask" is used with each coherence transaction to identify which processors should snoop that specific coherence action [56]. Martin *et al.* [57] proposed, *Timestamp snooping*, a technique that

processes address transactions in a logical order. The logical time order is achieved with some additional bits per address transaction and by performing a handshake at the network switches. Accordingly, total order is achieved at each processor or memory module by re-ordering transactions based on their timestamps. Martin *et al.* [14] later proposed a bandwidth-adaptive mechanism that can switch between a broadcast-based snooping protocol to unicast-based directory protocol based on the bandwidth usage information from the interconnect. Our bandwidth-adaptive hybrid protocol utilizes a similar mechanism for dynamic bandwidth utilization tracking, but our proposed protocol uses that information to adaptively decide whether to trigger update optimizations or not. More details on this bandwidth-adaptive protocol are given in Chapter 5 and Chapter 6.

Martin *et al.* [36] proposed a new cache coherence framework called *Token Coherence* that explicitly exchanges and counts tokens to establish coherence. Each cache line is associated with a fixed number of tokens equal to the number of processors in the system. A processor can read a cache line when it has at least a token for that specific line, and can only write when it has all the tokens for the line. In order to establish direct cache-to-cache transfers on an unordered interconnect, *Token Coherence* decouples the protocol implementation into correctness and performance substrates. As such, in the absence of a data race, fast approach is taken to achieve direct cache-to-cache transfers. However, in race conditions, correctness substrate becomes effective by issuing *persistent requests* to avoid starvation [36, 37]. Such persistent requests can become a performance bottleneck, and accordingly there were some studies that targeted token coherence optimizations [38, 58, 59, 60]. However, to the best of our knowledge there is still no commercial system that uses a token-based cache coherence protocol.

Directory-based protocols have been the choice of enforcing coherence when it comes to larger systems as they scale better than the other options. Examples of systems that use a directory-based approach for cache coherence include Stanford DASH [61], SGI

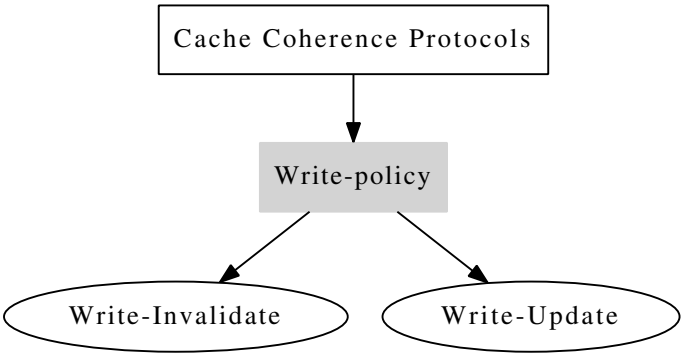


Figure 2.2: Classification of cache coherence protocols based on write-policy to shared data

Origin [62], Alpha 21364 [63], HP AlphaServer GS320 [64], IBM Power [51, 52], and Sun UltraSparc servers [65, 66]. The trend of putting more and more cores onto a single chip makes scalability an imperative design constraint especially for future CMPs. Hence, in this work directory scheme is utilized to implement the proposed cache coherence protocols mainly for scalability advantages over broadcast-based protocols.

2.2 Adaptive/Hybrid Cache Coherence Protocols

There have been numerous efforts to improve cache coherence protocols performance. The selection of the cache coherence write policy on shared data as depicted in Figure 2.2, write-invalidate versus write-update, has been the main focus of many earlier studies [33, 67, 68, 69, 70, 71, 72, 73, 74]. Karlin *et al.* proposed a hybrid update-invalidate protocol for bus-based systems, called *competitive snooping* [67]. This proposed protocol works as a write-update protocol and switches to invalidation of a cache block when after a number of updates, identified by the *competitive threshold*, there is no intervening local access. The scheme includes a counter with each cache line that is preset to the competitive threshold each time the block is accessed locally and decremented when the block is updated by another processor. The local copy is invalidated when the counter reaches zero, and updates end when all the copies are being invalidated. Eggers and Katz [75]

showed that competitive snooping improved performance of some applications but they did not perform well for applications exhibiting sequential sharing or also referred to as migratory sharing. Migratory sharing is characterized by exclusive read and write access by one processor at a time to a shared cache block.

Directory-based protocols induce longer latencies due to indirection through the directory, thereby latency reduction is more important for directory-based protocols compared to snooping protocols. Nilsson *et al.* [70] and Grahn *et al.* [34] studied the performance of a directory-based hybrid protocol, called *competitive-update*, that uses relaxed memory model to overlap write latency with local processing. Performance study showed lower miss rates for all applications tested, but some applications could not translate this into a performance improvement in the execution time because of an increased network traffic under competitive update. Accordingly, they proposed to extend the competitive-update protocol with a migratory data detection mechanism to reduce the network traffic. Dahlgren *et al.* further optimized such hybrid protocols with implementing a *write-cache* and providing read-snarfing [71, 72]. Read-snarfing is typically implemented in bus-based systems. In a system that implements *read-snarfing*, a data block that is transferred on the bus as a read response not only updates the processor that requested it, but also updates all other caches that have the same block invalidated. Anderson *et al.* [73] proposed to utilize a new metric, *block write run*, while designing hybrid protocols. Their study is also based on a bus based snoopy protocol which implements read-snarfing . Block write run is defined as a sequence of write references to a shared cache line, uninterrupted by either an access to that line by another processor, or replacement of the cache line [76]. In general, write-update based protocols perform better for short write runs, while long write runs favor write-invalidate based protocols. Accordingly, they used this write-run concept to vary the invalidate threshold in the competitive-update protocol.

Many papers have examined shared memory behavior and tried to optimize underlying

cache coherence protocols for specific static data sharing patterns such as migratory sharing [68, 69], read-modify-write sequences [74, 77], pairwise-sharing [78], and producer-consumer sharing [35, 79]. Cox *et al.* and Stenstrom *et al.* proposed an adaptive protocol that detects such migratory sharing data blocks in order to reduce the cost of moving them between processors. They optimized the base write-invalidate protocols by merging the read-miss requests with invalidation messages for the detected migratory objects. For other blocks, the protocol works as an ordinary write-invalidate protocol. On the other hand, in producer-consumer sharing, a subset of threads accesses a shared data item, one of which modifies the data while the others read each modification. More details about producer-consumer sharing can be found in Section 5.2.2. Cheng *et al.* [35, 79] studied such sharing pattern and corresponding prediction mechanisms in the context of ccNUMA shared memory multiprocessors. Their study inspired us to target the producer-consumer sharing pattern as well. However, their study entails off-chip memory accesses for each write-update which is a big performance penalty in CMPs. This thesis targets optimizations for CMPs and proposes extensions to satisfy varying bandwidth and latency characteristics of the running workloads. As such, not only we propose an adaptive protocol optimized for producer-consumer sharing patterns but also optimized based on bandwidth and latency awareness. More details about the level of improvements this thesis provides as compared to the protocol by Cheng *et al.* can be found in Chapter 5.

2.3 Cache Coherence Predictors

Another approach to deal with cache coherence overhead is based on coherence predictors. Mukherjee and Hill [80] proposed Cosmos coherence message predictor with an extended study on Yeh and Patt's two-level PAp branch predictor [81]. Kaxiras *et al.* [82, 83] proposed instruction-based predictors as an alternative to address-based predictors to move the

shared data close to the consumers as soon as possible. Lai *et al.* [84] proposed pattern-based memory sharing predictors that reduce the memory requirements of the previous proposed predictors. Acacio *et al.* [16] devised a prediction scheme for owner prediction to convert 3-hop cache-to-cache misses to 2-hop misses and they utilized prediction schemes to tackle upgrade misses in [85]. Upgrade misses occur when a store instruction is issued to a read-only copy of the line in the cache [85]. Coherence predictor cache was designed and evaluated in [86] as a resource-efficient coherence predictor. Nilsson *et al.* made an observation based on SPLASH-2 kernels that coherence activity footprint is confined to a small fraction of the whole data set. Further, they utilized address filtering to avoid unnecessary caching to the Coherence Predictor Cache which inspired us to use the same strategy for our Producer-Consumer Predictor Cache (PCPC). Martin *et al.* [87] analyzed commercial workloads and according to the sharing patterns of the observed workloads they proposed destination-set prediction to reduce indirections caused by directory protocols. Perceptron-based coherence predictors were suggested and experimented with in [88, 89]. Most of these extensions require major modifications and are relatively costly for a CMP design. Accordingly, we propose modest hardware extensions to implement our adaptive cache coherence protocols. More details about our prediction mechanism will be given in 5.2.2.

2.4 Producer-Consumer Communication

Producer-initiated communication was suggested to alleviate long miss latencies in shared-memory multiprocessors. Koufaty *et al.* studied data forwarding mechanisms utilizing compilers [90]. They proposed a framework in which when a processor produces a datum, in addition to updating its cache, it sends a copy of the datum to the caches of the processors that the compiler identified as consumers of it. Abdel-Shafi *et al.* evaluated fine-grain producer-initiated communication in cache-coherent multiprocessors [91]. They

investigated whether producer-initiated communication primitives can provide complementary benefits to software prefetching. They utilized two such software-controlled fine-grain primitives, *WriteSend* and *WriteThrough* also referred to as *remote writes* in their study. Byrd *et al.* conducted a performance analysis study on producer-initiated communication mechanisms in distributed shared memory (DSM) multiprocessors [92]. All these studies were at the software level. Our study looks at the problem, however from a hardware perspective and implements the proposed mechanisms as a hardware-based cache coherence protocol.

2.5 Cache Coherence Optimizations for Chip Multiprocessors

Cache coherence has been investigated heavily in the research community, but most of the prior work targeted multiprocessor machines (MPs) comprised of multiple single-core processors. With the advent of CMPs, the cache coherence problem becomes a first-order design issue at the chip level. Many constraints including interconnect, latency, power, and energy differ in MP from CMP designs.

Recently, there have been some studies targeting to optimize cache coherence protocols for CMPs. Cheng *et al.* proposed a heterogeneous interconnect for CMPs composed of wires with varying latency, bandwidth, and energy characteristics. Then, they advocated intelligently mapping various cache coherence operations to the appropriate wires. This work is orthogonal to our proposed optimizations and can be further utilized to optimize the performance of cache-to-cache updates [31]. Our work is based on homogeneous interconnects. Atoofian *et al.* utilized prediction to convert broadcast snooping messages to multicast messages. As such, they targeted reducing power dissipation in ordered networks

used by snooping protocols [93].

As more and more cores are integrated into a single chip, and also with the utilization of multiple CMP based systems, remote miss latencies will become a major performance issue in CMPs. Accordingly, Brown *et al.* proposed a proximity-aware directory-based cache coherence protocol optimized to alleviate the remote miss latency overhead. The proposed proximity-aware coherence protocol extends the current write-invalidate protocols by letting closest sharer to provide the missed cache line instead of the home node [24]. Again this work is orthogonal to our proposed optimizations. However, we propose to extend our adaptive cache coherence protocol with proximity awareness inside the chip. Direct cache coherence, DiCo-CMP, is suggested by Ros *et al.* as an alternative directory-based cache coherence protocols [94]. The idea is to distribute the directory responsibilities to the caches that can provide the cache block on a miss. Our proposed study assumes a directory-based protocol but it can be easily extended to incorporate DiCo-CMP.

Chapter 3

Evaluation Strategies and Testbeds

This chapter discusses the strategies for performance evaluations used to guide the proposed cache coherence protocols and assess the effectiveness of these solutions. As such, this chapter identifies the testbeds, benchmarks, simulators, and analyses used for evaluations. Hardware testbeds cannot reveal all the detailed information needed for understanding the performance impact of the cache coherence overhead and effectiveness of current solutions. Therefore, a number of simulators were used. On the other hand, simulators take orders of magnitude more time to execute similar tasks, hence, an analytical model will be considered for scalability studies. The workloads needed in each case as drivers, whether full applications or synthetic benchmarks, are identified in this chapter. Accordingly, this chapter first describes the evaluation plan. Then, we summarize the workloads used. Simulation strategies are discussed next, followed by a detailed discussion on the experimented testbeds. Finally, we give a brief summary on analytical analysis of cache coherence protocols which will be discussed in more details in Chapter 6.

3.1 Evaluation Methodology

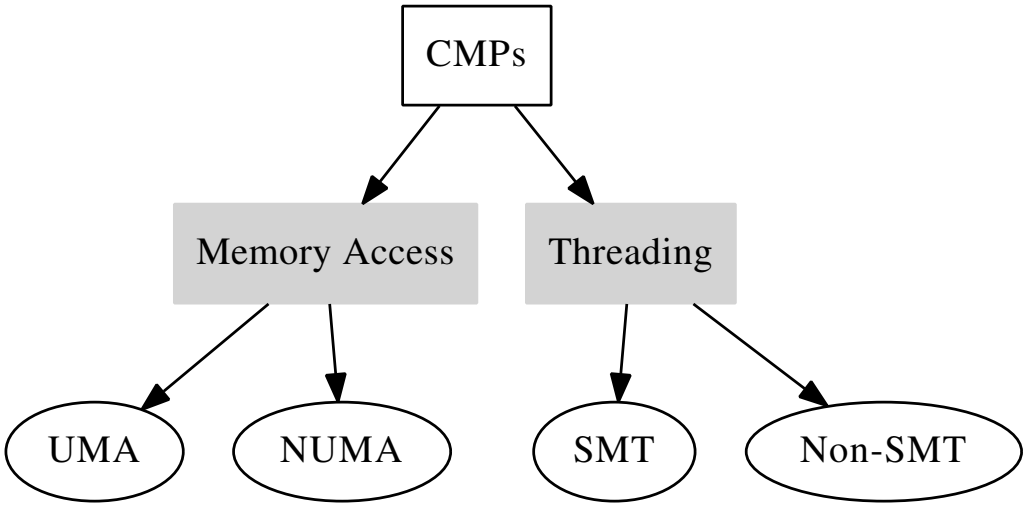


Figure 3.1: High-level classification of the experimented CMP systems

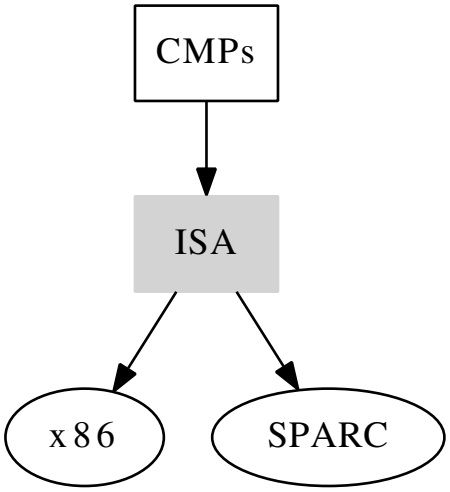


Figure 3.2: Experimented CMP systems based on the Instruction-Set Architecture (ISA)

In this section, we provide more details about our evaluation methodology. In doing so, we start with identifying the performance issues in emerging CMPs [95, 96]. By focusing on the major potential performance bottlenecks, we position the cache coherence impact with respect to the impact of the other performance aspects of CMPs. This work considers from single CMP systems to CMP-based clusters. As such, this chapter presents the experimented test beds in two categories. The testbeds are selected to represent the range of

Table 3.1: CMP specific performance analysis metrics

Focus of Analysis	Aspects	Metrics
Cache (via simulation)	- Overhead of maintaining cache coherency among the cores	- Coherence Overhead
Memory	- Bandwidth sustained from the cores to the main memory - Latency observed from the cores to the main memory	- Bandwidth - Latency
System - Cluster	- Effects of contention at the system level	- Speedup

CMP implementation options as much as possible. Therefore, the selected CMP testbeds include architectures offered by major chip-vendors including Intel, AMD, and Sun[1]. They provide a set of CMPs that have different memory access strategies, threading implementations, and instruction-set architectures. A high-level CMP classification considering these parameters can be found in Figure 3.1. Existing CMPs mostly follow the Uniform Memory Access (UMA) model similar to SMPs. Future chips however, will not be able to sustain this architectural model. As such, Non-Uniform Memory Access Model (NUMA) will become more popular mainly for its scalability advantages. This trend was also experienced in the earlier multi-processor systems. Therefore, a CMP-based cluster with two different cache-coherent node configurations was chosen in order to emulate large CMP-based cc-NUMA systems. This testbed gave us unique opportunities to demonstrate the cache coherence impact in the context of the overall system performance. Two different node architectures that differ mainly in size provide a platform to compare the cache coherence issues with the increasing number of processors. Such a system is crucial to explore as many future CMPs will have similar building blocks to reach many cores.

Table 3.1 provides an overview of this study by demonstrating the focus of analysis, observed aspects and the target CMP specific performance analysis metrics. All the experiments but the cache coherence overhead analysis were conducted on real systems.

Multiple-CMP based, a Sun cluster, is used to explore the performance issues that come

[1]Sun is acquired by Oracle.

with CMPs in a large scale system. By conducting a detailed performance evaluation study, we tried to observe the performance impact of the cache coherence overhead with respect to other major performance bottlenecks in CMPs. The experiments to explore the multiple-CMP based cluster were designed to isolate the performance bottlenecks and provide insight into possible solutions by improving the utilization of the resources. A set of scientific applications were utilized to examine the performance related to these two different node architectures. These applications are mostly part of the High Performance Computing Modernization Program's (HPCMP) Technology Insertion [97]. They include many research areas from weather modeling to molecular dynamics. Then, in order to interpret and understand these application performance results, various benchmarks were utilized. These benchmarks include some tests from the HPC Challenge (HPCC) Benchmarks [98] , LMbench [99] and also some synthetic micro-benchmarks that we developed to further examine the underlying performance issues. It will be shown how the architecture of the nodes affects the overall system performance. In addition, the effects of the underlying cache coherence protocol, the latency associated with the operating system services, and memory affinity will be provided. Furthermore, hardware counters were used by utilizing the Performance Application Programming Interface (PAPI) [100].

To test issues that are not possible on a real CMP system, simulation studies are conducted. In addition, considering the excessive time-to-solution limitations of simulators, an analytical performance model is utilized to assess the potential performance gains from our proposed cache coherence solutions for large CMPs. The analytical framework can be used to perform fast design space explorations to assess potential performance gains by flexibly changing application and architecture characteristics. The details of the analytical study will be presented in Chapter 5. Also, this chapter discusses the benchmarks and the applications used for performance evaluation. The corresponding results will be given in Chapter 4.

3.2 Experimental Workloads

This section explains the various benchmarks and applications used in our performance evaluation study.

3.2.1 Benchmarking Suites

Our evaluation approach recognizes the importance of the memory subsystem in performance of the multi-core architectures. Accordingly, low level micro-benchmarking experiments using HPC Challenge benchmark STREAM [101] and LMbench [99] were conducted to isolate the performance bottlenecks and do stress testing on the memory subsystem. All of the experiments were designed to be multithreaded using POSIX threads library. This helps us to examine the performance scalability with the increasing number of threads running simultaneously. All systems were using gcc compiler as can be seen in Table 3.4 with the corresponding version numbers. The x86 architectures were running Linux with couple of different kernels whereas UltraSPARC testbeds were all running Solaris 10 during this study. In order to alleviate system noise, the median values of 11 independent runs are reported as experimental results.

3.2.1.1 HPC Challenge Benchmarks

The HPC Challenge benchmarking suite includes seven different tests [98]. We used STREAM, FFT, and the effective bandwidth benchmark in our evaluation studies. STREAM benchmark is intended to test the sustained memory bandwidth of a system. STREAM measures the performance of four long vector operations using four loops. The first loop, COPY, measures transfer rates in the absence of any arithmetic operations; the second,

SCALE, multiplies a vector by a scalar value; the third operation, ADD, adds two vectors; and the last one, TRIAD, is a combination of SCALE and ADD operations. The sizes of the arrays must be defined to be larger than the last level cache of the evaluated system, so data are always fetched from memory. In addition, the code is structured with stride-one memory access to prevent data re-use in order to test only the memory bandwidth and all operations use 64-bit operands [101, 102]. However, we modified this test by reducing the array size to stay in cache and to examine the cache coherence overhead. Also, the STREAM benchmark was utilized to test the performance balance among the cores. FFT benchmark measures the floating point rate of computation of a one-dimensional Discrete Fourier Transform (DFT). The *effective bandwidth benchmark (b_eff)* is used to measure latency and bandwidth of various simultaneous communication patterns [98]. Message sizes, communication patterns and methods are used as the main parameters in the benchmark, and these parameters are tested for various combinations of values [103]. As a result, the logarithmic average of these variations is calculated as the effective bandwidth of the system.

3.2.1.2 LMbench

LMbench provides a suite of benchmarks to isolate the performance bottlenecks that are commonly found in many system applications. Micro-benchmarks were designed to measure system latency and bandwidth among processor and memory, network, file system, and disk [14]. For the scope of this study, memory bandwidth and latency tests were adapted for our experiments.

3.2.2 Applications

In order to test the system level performance, several scientific applications were used. Those include LAMMPS [104], HYCOM [105], NAMD [106], WRF [107], AMR [108], AVUS [109], CTH [109, 110], ICEPIC [110], OVERFLOW 2 [111], and ChaNGa [112]. LAMMPS is used to model an ensemble of particles in a liquid, solid or gaseous state [113]. HYCOM is a primitive equation ocean general circulation model that is based on the Miami Isopycnic-Coordinate Ocean Model (MICOM) [105]. NAMD is a parallel molecular dynamics code designed for high-performance simulation of large biomolecular systems [106]. WRF is a next-generation mesocale numerical weather prediction system developed to support both operational forecasting and atmospheric research needs [107]. AMR is a gas dynamics code built upon the Adaptive Mesh Refinement framework from Lawrence Berkeley National Laboratory [108]. AVUS is a Computational Fluid Dynamics (CFD) code designed to determine the fluid flow and turbulence generated by air vehicles [109]. CTH is an Eulerian finite-volume code for modeling complex multidimensional problems that involve shock wave propagation, multiple materials, and large deformations [109, 110]. ICEPIC solves Maxwell's equations and Lorentz force law [114]. OVERFLOW 2 is another CFD code that is a flow solver for modeling compressible, viscous flow [111]. ChaNGa is performing collision-less N-body simulations [112].

Most of these applications (LAMMPS, HYCOM, NAMD, WRF, AMR, AVUS, CTH, ICEPIC and OVERFLOW 2) have been used as benchmark codes for evaluating vendor offerings to the DoD High Performance Computing Modernization Program (HPCMP) [97, 115]. They are proxies for the applications that are important to HPCMP and represent a significant portion of the workload of the program. As a suite, they represent not only major application areas but also major algorithmic approaches and performance characteristics (light or heavy I/O or communication, different programming models, etc.). Therefore, the

performance of this collection is an important metric to at least this one segment of the scientific computing community. ChaNGa was included for a different reason, namely its unique performance properties owing to its underlying computational model (asynchronous and message-driven). That model differs from the regular, grid-based model typical of scientific codes, including the other codes in our suite, with the exception of NAMD. ChaNGa is based upon the Charm++ system [116] (as is NAMD) and its performance characteristics are the topic of more discussion later in Section 4.2.7.

3.2.3 Simulation Workloads

Table 3.2: Simulation Workloads

Benchmark	Description	Input Data Set
barnes	Implements the Barnes-Hut method to simulate the interaction of a system of bodies (N-body problem)	16384 particles
cholesky	Blocked Cholesky Factorization on a sparse matrix	input tk29.O
fft	Complex 1-D, 6-step FFT	64K complex data points
ocean	Simulates large-scale ocean movements based on eddy and boundary currents	258*258 array

The benchmarks for the full-system simulation study were selected from the SPLASH-2 benchmark suite [117]. More details on the selected benchmarks including the input data sets are provided in Table 3.2.

Barnes is an implementation of the Barnes-Hut method to simulate the interaction of a system of bodies over time in a gravitational system (N-body problem). Cholesky kernel performs a blocked cholesky factorization on a sparse matrix. FFT benchmark solves a complex, one-dimensional version of the $radix - \sqrt{n}$ six-step FFT. Ocean application is a simulation of large-scale ocean movements based on eddy and boundary currents.

Table 3.3: Simulation Parameters

Parameter	Value
Processors	In-order 2-way UltraSPARC-III Cu processor
# of cores	2,4,8,16 core target systems
L1 Caches	Split I&D, 64KB 4-way set associative with 64-byte blocks per core
L2 Caches	1MB 4-way set associative with 64-byte blocks per core
Coherence Mechanism	snooping protocol based on MOSI states
Memory	256MB per core shared memory
Interconnect	Hierarchical Switch

3.3 Simulation Strategies

In addition to the benchmarking experiments, we also conducted a simulation study to investigate low-level cache performance issues mainly for two reasons. First, the simulation study gives us the opportunity to investigate currently non-existing hardware. Second, the goal of the cache performance analysis study is to isolate the overhead of maintaining caches coherent inside the multi-core chips, and it is impossible to get statistics on cache state transitions on a real hardware. We present some initial results, timing measurements based on a synthetic benchmark, from cache coherence overhead analysis in our earlier work [95]. However, it is very hard to eliminate the overhead due to the operating system activities from the timing results. Thus, we decided to go with a simulation study in which various protocol messages and state transitions can be isolated to get an accurate measurement of coherence misses which can be used as a better metric to investigate cache coherence overhead.

Cache coherence overhead in multi-core systems is investigated with full-system simulation utilizing Virtutech Simics [27] extended with the GEMS toolset [28]. A full-system simulator was chosen to include operating system effects to establish a more practical platform for our measurements. Simics provides the functional full-system simulation and GEMS provides a detailed memory system timing infrastructure which observes all cache

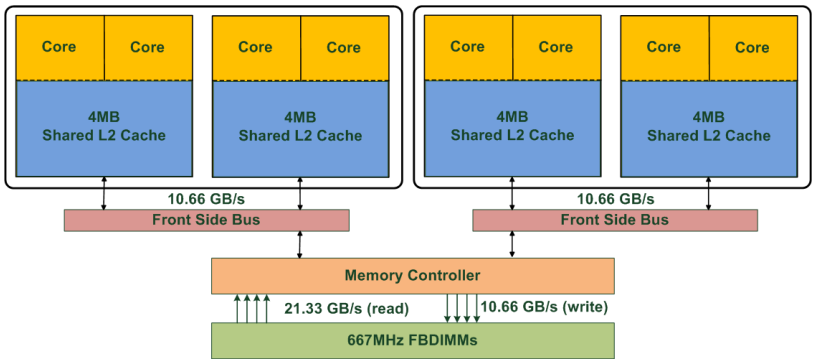
coherence protocol messages and the corresponding state transitions.

Solaris 10 is installed in all target systems including 2-core, 4-core, 8-core, and 16-core SPARC machines. Table 3.3 provides the details of the simulation environment. The cache coherence protocol utilized in our testbeds is a snooping protocol with MOSI states. The reason behind the snooping protocol selection is the fact that our real CMP hardwares are mostly using such a protocol. Moreover, we will provide comparative results for directory based protocols in Chapter 6, as our proposed adaptive protocols are all based on a directory scheme. In order to simulate only the computational part, breakpoints are inserted into the benchmarks and checkpointing is used. Furthermore, caches are warmed up to avoid cold misses as suggested in [118].

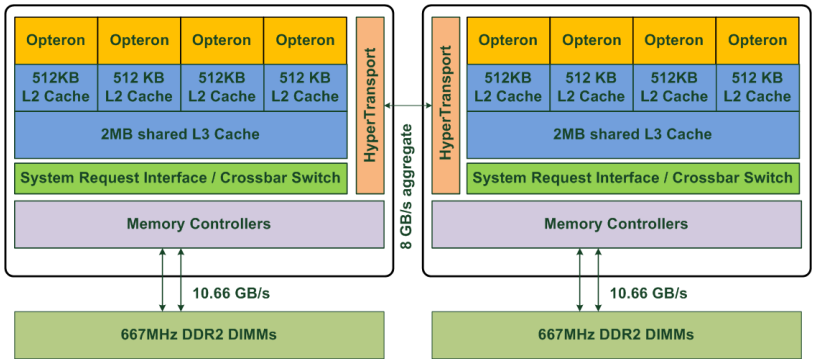
3.4 Testbeds

Table 3.4: System Characteristics of the Experimental Testbed

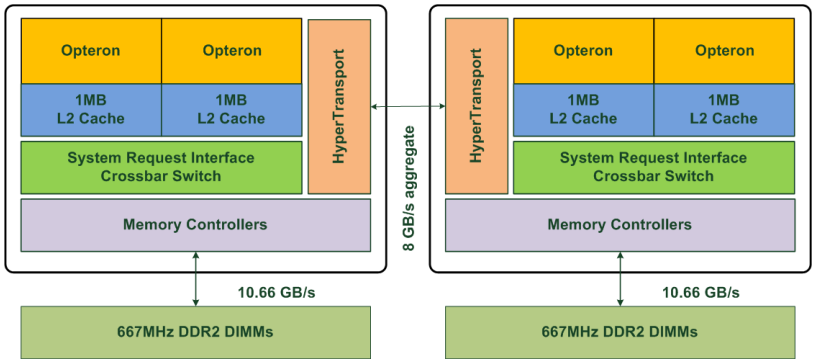
System	Intel Core2 E6000	Intel Core2 Q6600	Intel Xeon E5310	SunFire X2200 M2 Server	SunFire X2200 M2 Server	SunFire T1000 Server	SunSPARC T5140 Server
Processor	Intel Core2 (Conroe)	Intel Core2 (Kentsfield)	Intel Xeon (Clovertown)	AMD Opteron 2218 (Santa Rosa)	AMD Opteron 2356 (Barcelona)	UltraSPARC T1 (Niagara)	UltraSPARC T2+ (Victoria Falls)
Clock (GHz)	2.4	2.4	1.66	2.6	2.3	1.0	1.2
L1 Data Cache	32KB private	32KB private	32KB private	64KB private	64KB private	8KB private	8KB private
L2 Cache (per socket)	4MB shared	2x4MB shared by 2	2x4MB private	2x1MB private	4x512KB private	1x3MB shared	1x4MB shared
L3 Cache (per socket)	-	-	-	-	2MB shared	-	-
Sockets	1	1	2	2	2	1	2
Cores/socket	2	4	4	2	4	6	8
Memory Access	UMA	UMA	UMA	NUMA	NUMA	UMA	UMA
Threading	Non-SMT	Non-SMT	Non-SMT	Non-SMT	Non-SMT	SMT	SMT
Threading Library	pthreads	pthreads	pthreads	pthreads	pthreads	pthreads	pthreads
HW Threads	-	-	-	-	-	4 per core	8 per core
ISA	x86	x86	x86	x86	x86	SPARC	SPARC
Compiler	gcc 4.3	gcc 4.3	gcc 3.4	gcc 4.2	gcc 4.3	gcc 4.0	gcc 4.0
OS	Linux 2.6.27	Linux 2.6.27	Linux 2.6.9smp	Linux 2.6.5smp	Linux 2.6.27	Solaris 10	Solaris 10



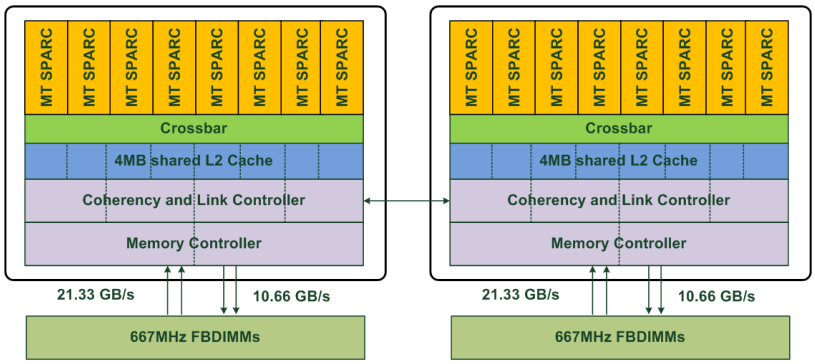
(a): Dual-socket Intel Xeon E5310 (Clovertown)



(b): Dual-socket AMD Opteron 2356 (Barcelona)



(c): Dual-socket AMD Opteron 2218 (Santa Rosa)



(d): Dual-socket UltraSPARCT2+ (Victoria Falls)

Figure 3.3: Architectural overview of some of the experimental testbeds

This section briefly describes the experimental testbeds used in this study. Figure 3.1 and Figure 3.2 shows a high-level classification of CMPs offered from different vendors mainly representing the test-beds used in this work. Table 3.4 provides a summary of key architectural and system features. To analyze the emerging homogeneous multi-core architectures, a broad range of systems were utilized. Our experimental testbeds include processors from most of the major microprocessor vendors with different examples of cache-based x86 and UltraSPARC architectures. Intel based testbeds follow a Symmetric Multi-Processor (SMP) architecture, with a shared Front-Side Bus (FSB) connecting the processors to a single controller, called a north bridge. On the other hand, AMD64 Opteron based testbeds are cache-coherent Non Uniform Memory Access (cc-NUMA) architectures. Each socket/chip contains its own memory while the whole memory in the system is globally shared by utilizing the HyperTransport (HT) links. NUMA comes from the fact that different memory parts can be accessed with different latencies in such architectures. In addition, AMD64 Opteron systems have the memory controller integrated into the processor with direct-connect architecture [119]. This can be an advantage for two reasons as explained in [102]. First, the memory controller operates at the same frequency as the core. So, as the core speed is increased, the memory controller speed is also increased, reducing the latency through the memory controller, allowing faster access to memory. Second, scalability is sustained by adding more paths to memory when the number of cores is increased, which satisfies the demand for higher memory bandwidth. More details will be given for each system in the following subsections. Architectural overview of some of the evaluated platforms can be found in Figure 3.3.

3.4.1 Intel Core 2 Duo E6600 (Conroe)

Conroe is the successor of Intel Core Duo desktop processor line. It is based on 65-nanometer (nm) silicon technology and it improves the performance while keeping the

power consumption low compared to its predecessors. Each core runs at 2.4 GHz with 1066 MHz front side bus (FSB) and can complete four full instructions simultaneously by utilizing a 14-stage pipeline. In addition, cores can issue one 128-bit Streaming SIMD Extension (SSE) instruction per cycle. In fact, Conroe uses two floating point calculation units, one dedicated to addition and the other to multiplication and division. The combined operation of multiplication and addition exploits the two units conjointly and is executed with a transfer rate of one cycle for the two operations, in other words two 128-bit operations per cycle which yields to a peak double-precision performance of 9.6 GFlop/s per core [120].

The L1 instruction and data caches have 32KB capacity and they are 8-way set associative with a 64-byte cache line size. Each core has two private L1 caches and they link with a 16-way set associative shared L2 cache of size 4MB. The cache line size for L2 cache is also 64 bytes. This work evaluates a single socket Conroe based system.

3.4.2 Intel Core 2 Quad Q6600 (Kentsfield)

Kentsfield or Intel Core 2 Quad Q6600 processor as suggested from its name is actually a combination of previously mentioned two Core 2 Duo E6600 (Conroe) processors onto a single multi-chip module (MCM). Accordingly, it has almost the same features as Conroe. However, MCM contains two dies that are physically independent of each other. That is, there is no internal data sharing. As such, L2 cache is now shared only by half of the cores (two cores) available, creating a 2x4MB L2 cache structure. This also means that Advanced Smart Cache features of Intel Core 2 Duo processors, dynamically allocate shared L2 cache based on workload, cannot be fully utilized among the four cores. In addition, any communication between the cores on different dies needs to go through the slower FSB which can be a bottleneck for memory intensive applications. This work evaluates a single

socket Kentsfield based system.

3.4.3 Intel Xeon E5310 (Clovertown)

Clovertown similar to Kentsfield follows the same design methodology that combines two Intel Core 2 dies into a single MCM. Clovertown suffers from the same bottlenecks as mentioned earlier for Kentsfield and shares the same core architectural features with Conroe. On the other hand, our Clovertown testbed runs at 1.66 GHz and is a dual-socket platform as can be seen in Figure 3.3a.

3.4.4 AMD Opteron 2218 (Santa Rosa)

This study evaluates the Sun Fire X2200 M2 dual-socket platform based on Santa Rosa processors with a clock rate of 2.6 GHz as can be seen in Figure 3.3c. Santa Rosa is a dual core AMD64 Opteron processor. It is based on 90-nanometer (nm) silicon technology. Each core has 64KB 2-way set associative L1 data and instruction caches and 1MB 16-way set associative L2 cache. Both the L1 and L2 caches are private and the cache coherency is maintained based on the MOESI (Modified, Owned, Exclusive, Shared, Invalid) protocol. A single cache-coherent HT link is used to connect the sockets. Each HT link is operating at a peak rate of 1GHz corresponding to a uni-directional bandwidth of 4GB/s [121]. AMD64 Opteron processors use 12-stage integer pipeline and 17-stage floating point pipeline.

3.4.5 AMD Opteron 2356 (Barcelona)

This study evaluates the Sun Fire X2200 M2 dual-socket platform based on Barcelona processors as can be seen in Figure 3.3b. Barcelona is a single-chip, native quad-core design

from AMD. It is manufactured using 65-nm silicon technology. Each core operates at 2.3 GHz and fully supports 128-bit floating point SSE instructions for peak double-precision performance of 9.2 GFlop/s per core. Cache hierarchy has been extended, compared to dual-core AMD64 Opteron processors. Barcelona features a cache controller and three stages of caches. The first two levels of caches are similar to aforementioned Santa Rosa processor. The difference comes with the new 2MB 32-way set associative shared L3 cache based on non-inclusive victim cache architecture. Two different prefetchers are utilized, core prefetcher and DRAM prefetcher. Core prefetcher places data in the L1 cache of the requesting core, whereas all DRAM prefetched data is placed into the L3 cache [122].

3.4.6 Sun UltraSPARC T1 (Niagara)

The UltraSPARC T1, also known as Niagara, is a multi-threading multi-core processor from Sun. Niagara follows a totally different design methodology compared to aforementioned x86 architectures, based on "Chip Multi-Threading (CMT)" technology. CMT encompasses the multi-core technology with hardware threads and "Simultaneous Multi-Threading (SMT)" to improve the instructions processed per cycle. Niagara cores are relatively less complex compared to other contemporary high-end processors in order to make space for eight cores on the same die. Niagara processor does not support out-of-order execution and only has one floating point unit (FPU). Therefore, this study only presents a subset of results from Niagara just for comparison purposes. Our Niagara testbed, Sun Fire T1000 server, has six cores and four hardware threads per core with a clock rate of 1.0GHz. All cores connected to memory and the I/O subsystem through a 134GB/s crossbar switch. The dedicated per core L1 data cache is 8KB and L1 instruction cache is 16KB. However, the 3MB L2 cache is shared among all the cores and hardware thread contexts. Sun improved this processor by adding more Arithmetic Logic Units (ALUs) and FPUs in its new releases, T2 and T2 plus.

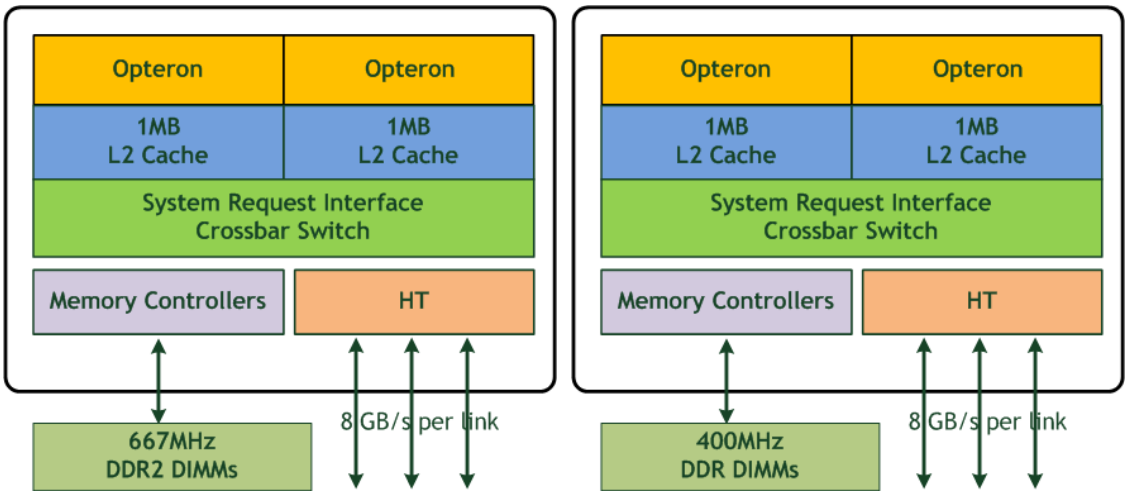
3.4.7 Sun UltraSPARC T2+ (Victoria Falls)

The Sun UltraSPARC T2 plus (T2+), also known as Victoria Falls, is an improved successor of T1 and T2 processors. T2+ can be seen as the multi-processor version of T2. In this study, we provide results from T2 plus. The architectural details can be seen in Figure 3.3d. Each Victoria Falls core has two groups of four hardware thread contexts providing a total of 64 simultaneous hardware threads per socket. Another improvement is the floating point capacity which is increased to one per core. This study examines the Sun UltraSPARC T5140 server with two T2+ processors operating at 1.2 GHz, with a per-core peak performance of 1.2 GFlop/s. The shared L2 cache is increased to 4MB. In addition, coherency links are added to support SMP.

3.4.8 Sun Fire Cluster (Midnight)

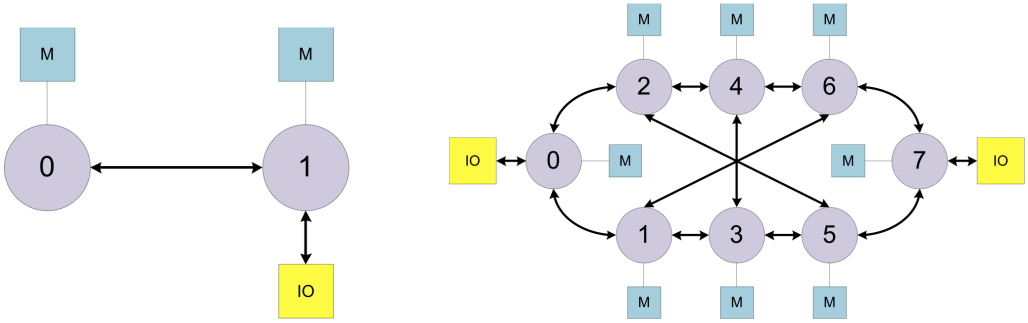
This testbed is used to analyze the cache coherence impact on multiple-CMP based systems. The main study was conducted on a 2312 compute cores Sun cluster, Midnight, based on Sun Fire X2200/X4600 servers at the Arctic Region Supercomputing Center (ARSC). As the machine uses two different compute node servers, the performance behavior of the system was characterized particularly in relation to the node level memory hierarchy performance using the two different node architectures based on the number of sockets: thin nodes and thick nodes. Thin nodes represent the two socket Sun Fire X2200 M2 servers and thick nodes refer to the eight socket Sun Fire X4600 servers. Corresponding node architectures are depicted in Figure 3.4.

In order to eliminate terminology conflicts, we define a number of the terms used. A socket/chip contains two cores (dual-core AMD64 Opteron) and the links to outside. A core is the fundamental execution unit in the system, having dedicated L1 and L2 level caches



(a): Sun Fire X2200 core architecture

(b): Sun Fire X4600 core architecture



(c): Sun Fire X2200 node architecture (thin node)

(d): Sun Fire X4600 node architecture (thick node)

Figure 3.4: Architectural overview of thin and thick nodes in Midnight cluster

but sharing the local memory with the other core on the same die. A node is then a group of sockets/chips (2 *or* 8 in our case) that share main memory and use the HyperTransport links to establish the intra-node communication. Finally, the system is a cluster with multiple nodes connected by an Infiniband network.

Table 3.5 describes the architectural features of the node architectures and the system features of the cluster. The experimental testbed is a Sun cluster based on Sun Fire X2200 and Sun Fire X4600 servers. This cluster is comprised of 415 shared memory nodes with a total of 2312 compute cores. The system includes 358 Sun Fire X2200 compute nodes each with

Table 3.5: System characteristics of the experimental testbed - Sun Fire cluster

Name (as referred to in this study)	**Thin nodes**	**Thick nodes**
Node Architecture	Sun Fire X2200 M2 Server	Sun Fire X4600 Server
# of nodes	358	55
Sockets/node	2	8
Cores/socket	2	2
Total number of cores	1432	880
Processor used	Dual-core AMD64 Opteron 2218(Santa Rosa)	Dual-core AMD64 Opteron 885(Egypt)
Core clock frequency (GHz)	2.6	2.6
Floating point/clock/core	2	2
Peak Performance / core (GFlop/s)	5.2	5.2
Technology (nm)	90	90
L1 Cache Size (KB)	64 (Instruction) & 64 (Data)	64 (Instruction) & 64 (Data)
L2 Cache Size (MB)	1	1
L2 Cache Type	16-way set associative, private	16-way set associative, private
Local memory/core (GB)	4	4
Memory Type	DDR2-667MHz	DDR-400MHz
Local memory/node (GB)	16	64
Intra-node communication	HyperTransport (HT)	HyperTransport (HT)
Frequency of HT (GHz)	1	1
Transfer rate of HT/link (GB/s)	8	8
# of HT links/socket	3	3
Operating System	SuSE Linux Enterprise Server 9.3 with kernel 2.6.5-7.276-smp	SuSE Linux Enterprise Server 9.3 with kernel 2.6.5-7.276-smp
Compilers	gcc v3.x, PathScale	gcc v3.x, PathScale
Interconnect	Voltaire Infiniband	Voltaire Infiniband
File System for the cluster	68TB Lustre	

16 GB of shared memory (4 GB per core), two dual core 2.6 GHz AMD Opteron processors and one 4X Infiniband Network Card on PCI-Express Bus per node. In addition, the system also has 55 Sun Fire X4600 compute nodes each with 64 GB of shared memory (4 GB per core), eight dual core 2.6 GHz AMD Opteron Processors and one 4X Infiniband Network Card on PCI-Express Bus per node. AMD Opteron processors that are used in these nodes have the same architectural features including 64KB two-way associative L1 cache and 1MB 16-way associative L2 cache. The X2200 server nodes are version M2 which use DDR2-667MHz memory whereas the thick nodes are using DDR-400MHz memory. The system also has two Sun Fire X4600 nodes which act as login nodes each with 32 GB of

shared memory (4 GB per core), four dual core AMD Opteron Processors and one 4X Infiniband Network Card on a PCI-Express Bus. The corresponding architectures are shown in Figure 3.4.

The sockets are represented in circles with their corresponding numbers in Figure 3.4c and Figure 3.4d. In addition, the arrows in Figure 3.4c and Figure 3.4d show the HyperTransport links that are used to connect the processor chips (sockets) inside the nodes. Each HyperTransport link operates at a peak rate of 1GHz yielding a bi-directional bandwidth of 8GB/s [123]. Each processor chip (dual-core AMD64 Opteron) has its own local memory and all the remote memory locations within the node are accessible using the HyperTransport links in the corresponding topology, which provides a coherent shared address space in hardware within the node. In order to maintain the coherency in the memory hierarchy, the MOESI (Modified, Owned, Exclusive, Shared, Invalid) cache coherence protocol is used in these AMD64 architectures. Details will be discussed in Section 4.2.1.

3.5 Analytical Models

Analytical prediction of performance is important as we see more and more cores are integrated onto CMPs. Simulators would take orders of magnitude more time to obtain results. As such, an analytical model to assess the potential performance gains from our proposed adaptive protocols is considered as a fast tool to explore various design options. In particular, such a tool is crucial for scalability studies due to time and infrastructure limitations from simulators.

Chapter 4

Cache Coherence Impacts in Modern Architectures: An Empirical Study

In this chapter, we present the results from our performance characterization work. The results were obtained according to the evaluation strategies and the testbeds discussed earlier in Chapter 3. The goal of this performance characterization study is to provide a better understanding of the performance impact of the cache coherence overhead in modern architectures. The study has also provided some performance optimizations to alleviate the cache coherence overhead in a multi-CMP ccNUMA system as a case study, which are not discussed here but more details can be found in [22, 121, 124]. The performance results from this study confirm the motivations for this research work, they show that the overhead associated with cache coherence mechanisms can over-shadow the performance gains from utilizing more cores in CMP-based systems.

The results are provided in two main sections. We first provide results from CMP architectures that are populated with one or two sockets. In such systems, cache-to-cache latencies are relatively shorter, and other performance aspects due to contention in shared resources

become more important. Therefore, results from these systems focus on such performance issues including memory bandwidth and latency. On the other hand, we also provide results from a multi-CMP system that generally requires more hops to travel between various cores in the system. As such, cache-to-cache latencies become more evident in such architectures. With the network traffic overhead that comes with the cache coherence mechanisms, these latencies can become the main performance bottleneck. Thereby, this testbed is further utilized to emulate a larger CMP with multiple hops of latency between various cores. Our results and analyses focus on characterizing the performance impact of the cache coherence mechanism by providing results from various parts of the system. Accordingly, Section 4.1 discusses the performance results from relatively smaller CMP systems, mainly at the chip-level that have one or two sockets and Section 4.2 presents the results from a multi-CMP cc-NUMA system. This multi-CMP system includes two different node architectures that differ in size. This helps us to make comparative analysis on cache coherence overhead with the increasing number of cores in the system. Furthermore, we also provide results from simulators that present cache coherence overhead analysis that cannot be otherwise obtained from hardware testbeds.

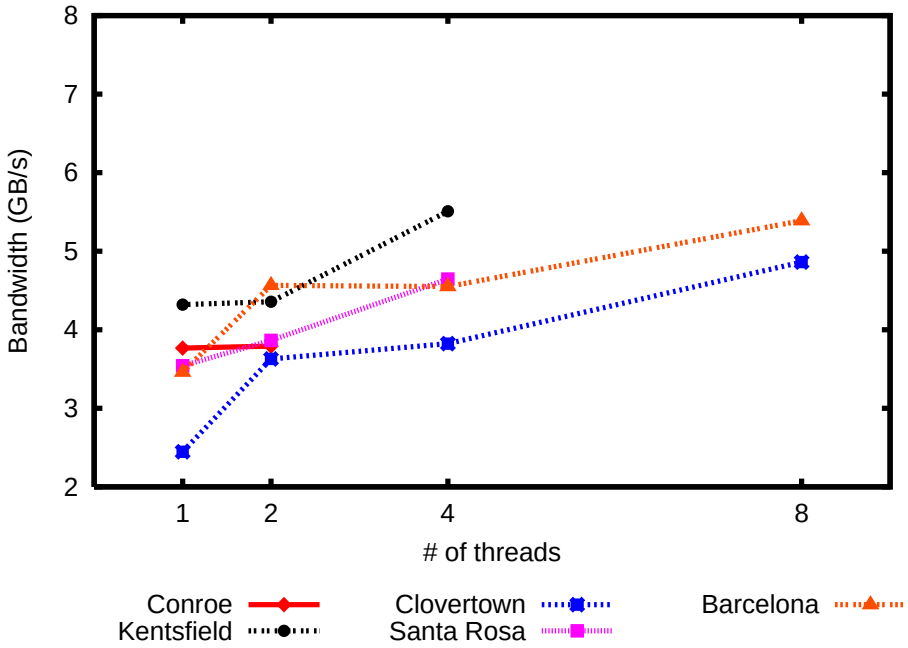
4.1 Experimental Results From Emerging CMPs

In this section, a broad range of homogeneous CMP based systems are investigated in terms of CMP specific essential performance behaviors. For the evaluated CMP systems, individual results are provided along with the comparative analysis considering the architectural and system features. To characterize the performance, we used micro-benchmarks from High Performance Computing Challenge (HPCC) [98] and LMbench [99] benchmarking suites. Performance analysis is conducted on CMPs from UltraSPARC and x86 architectures; including systems based on Conroe, Kentsfield, Clovertown, Santa Rosa, Barcelona,

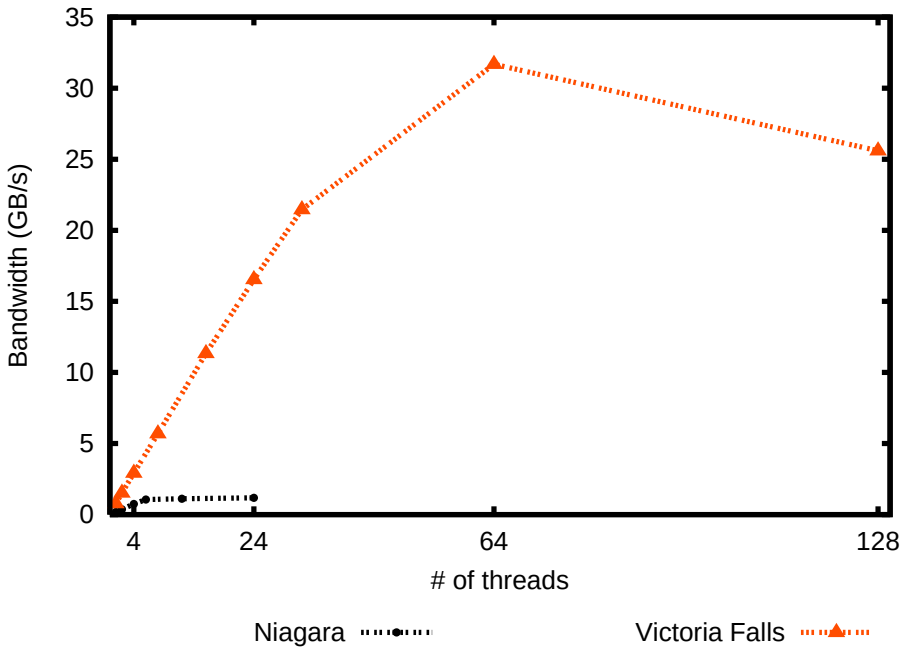
Niagara, and Victoria Falls processors. More details about these testbeds and our evaluation methodology were presented in Chapter 3. Finally, cache coherence overhead is analyzed using a full-system simulator. Experimental analysis and observations in this study aim to provide a better understanding of the emerging homogeneous CMPs.

4.1.1 Memory Bandwidth Analysis

For memory bandwidth experiments, STREAM benchmark and LMbench memory bandwidth tests were used. Figure 4.1 presents results from STREAM TRIAD benchmark. Figure 4.1a compares the results between x86 testbeds whereas Figure 4.1b shows results between UltraSPARC testbeds. For this test, the original STREAM benchmark is modified similar to StarSTREAM benchmark but with pthreads so that each thread runs its own copy of the benchmark independently. The goal is to evaluate the memory bandwidth using different array block operations with the increasing number of threads. As it is mentioned earlier, it is crucial to investigate the performance scaling of CMP systems in multi-threaded workloads to understand the actual efficiency and limits of such systems. Accordingly, it is observed that all x86 platforms are suffering with the increasing number of threads mainly due to the contention in the shared resources. Performance scaling behavior is similar between Clovertown and Barcelona although Barcelona outperforms Clovertown which should be mainly because of the difference in their clock rates. Santa Rosa scales relatively well and performs similar to Barcelona at four cores. On the other hand, Victoria Falls is really outperforming the rest of the testbeds with good scaling performance up to 64 threads and shows the results of the improvements in UltraSPARC architecture which can be seen by comparing it to the Niagara system. In addition, comparison of the peak STREAM results can be seen in Figure 4.2. From these results, it appears that efficient hardware support is needed to exploit thread-level parallelism to get performance gains from CMPs. For example, the internal crossbar network that connects the SPARC cores



(a): x86



(b): UltraSPARC

Figure 4.1: STREAM TRIAD Results

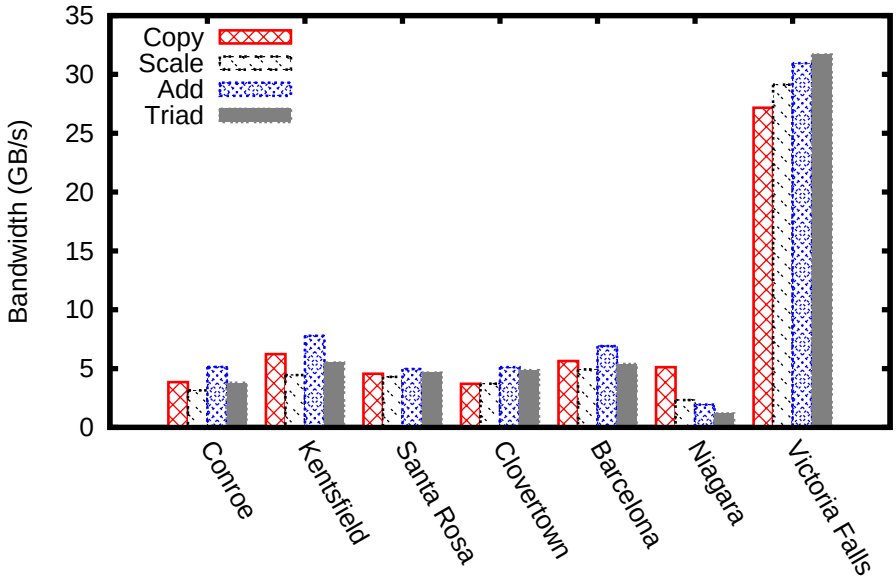
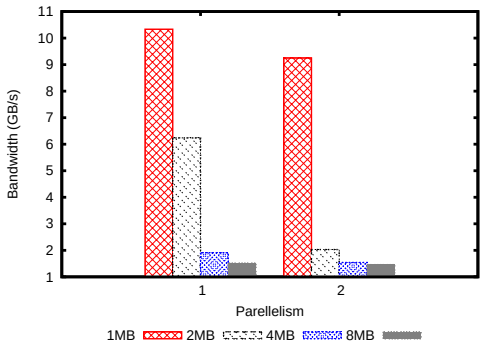


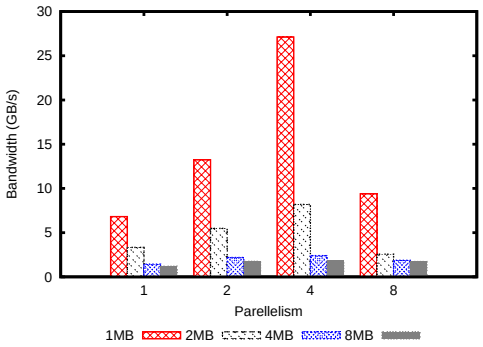
Figure 4.2: Comparison of peak STREAM Results

to L2 banks helped the Victoria falls processor to get better scalability results. Most of the scalability problems are observed when two or more cores are competing for a shared resource such as the memory controller or the shared bus. In addition, results from the UltraSPARC testbeds show the importance of having sufficient arithmetic units for all cores to achieve performance gains from thread-level parallelism (TLP).

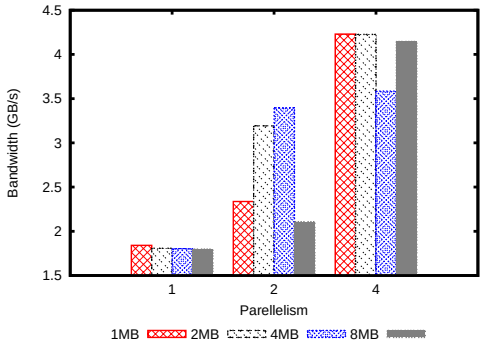
For the rest of the experiments, results from five of the testbeds are provided for brevity. Figure 4.3 illustrates the memory copy bandwidth results from LMbench. Memory copy operation was done between two allocated spaces of size 1MB, 2MB, 4MB, and 8MB. Results are provided for each system as well as the peak results for 8MB. Victoria Falls is again outperforming the rest by a margin and shows good scalability. Conroe is performing well on the cache level copy but for bigger sizes the bandwidth drops dramatically. Santa Rosa is second in terms of peak values and also shows relatively better scalability and



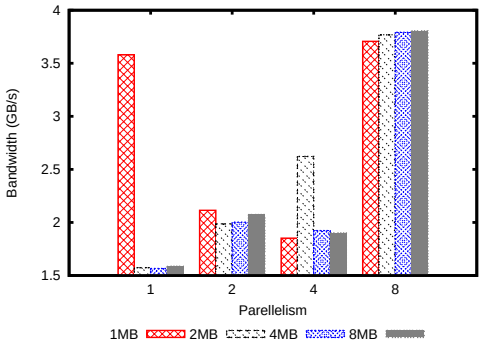
(a): Conroe



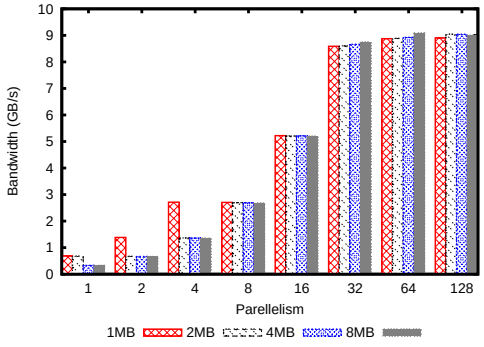
(b): Clovertown



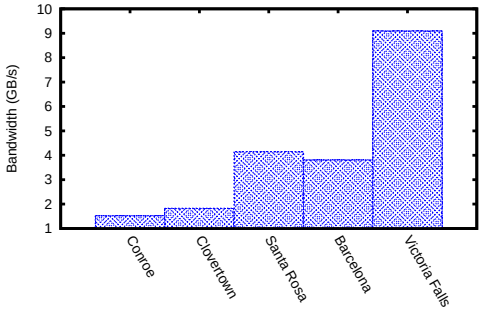
(c): Santa Rosa



(d): Barcelona

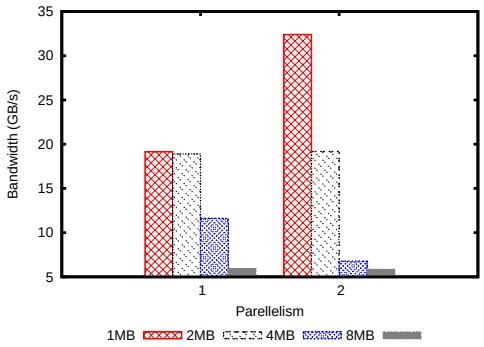


(e): Victoria Falls

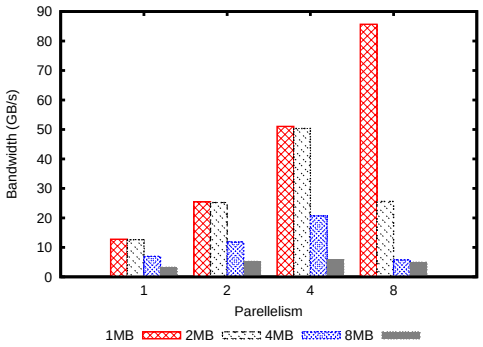


(f): Peak Results (8MB)

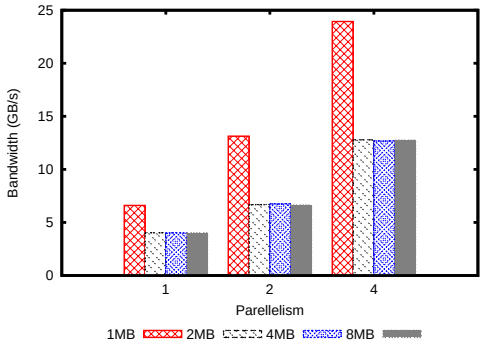
Figure 4.3: Memory Copy Bandwidth Results for Different Data Sizes



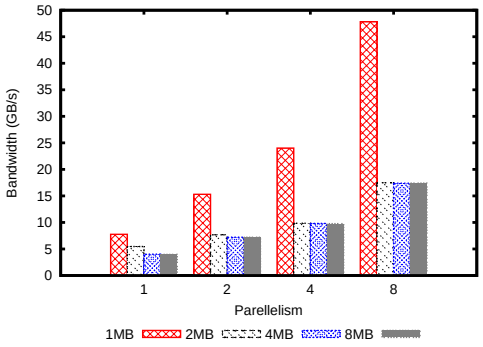
(a): Conroe



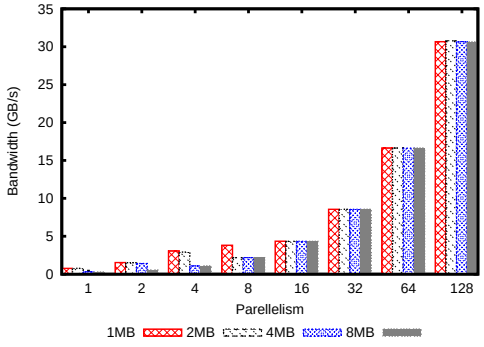
(b): Clovertown



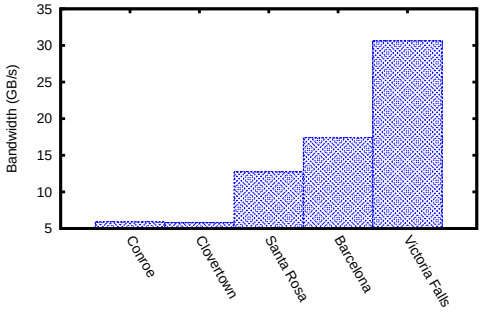
(c): Santa Rosa



(d): Barcelona

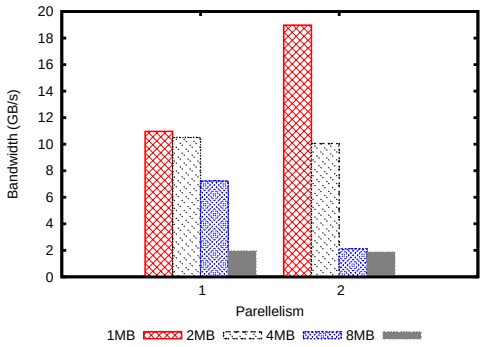


(e): Victoria Falls

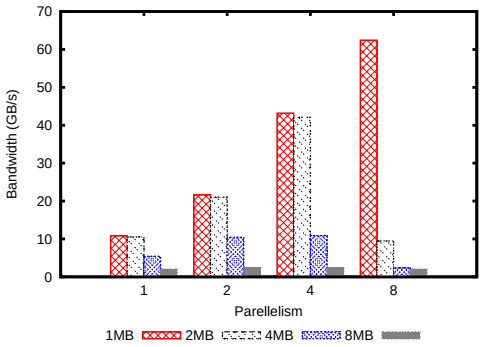


(f): Peak Results (8MB)

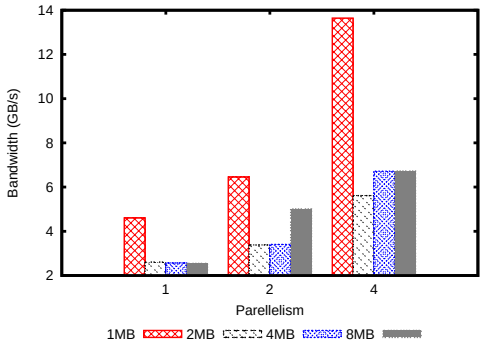
Figure 4.4: Memory Read Bandwidth Results for Different Data Sizes



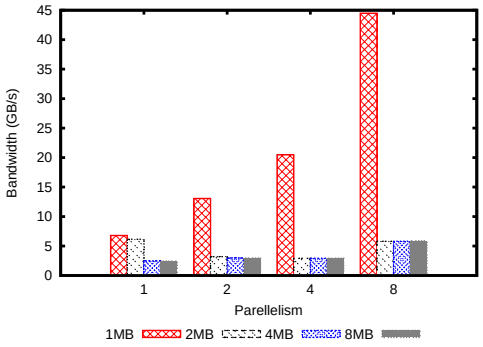
(a): Conroe



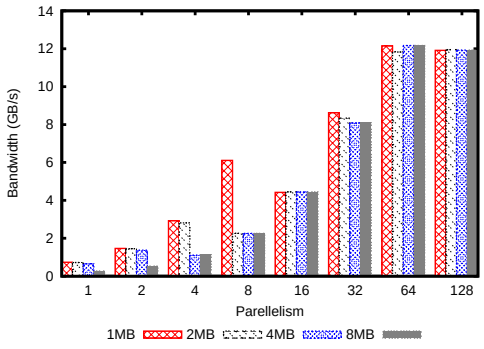
(b): Clovertown



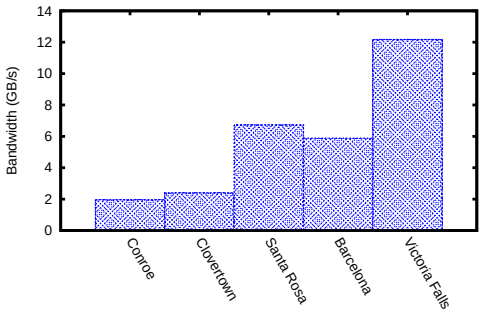
(c): Santa Rosa



(d): Barcelona

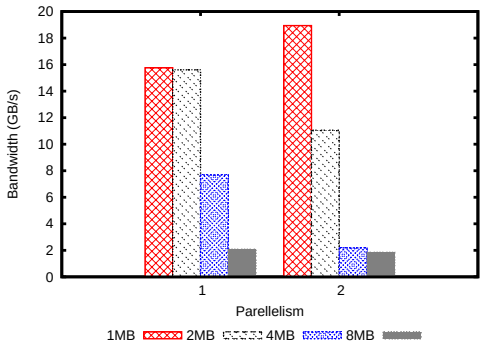


(e): Victoria Falls

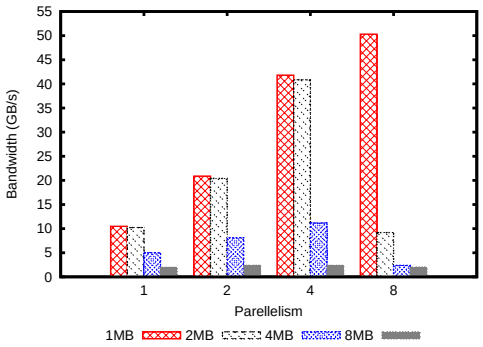


(f): Peak Results (8MB)

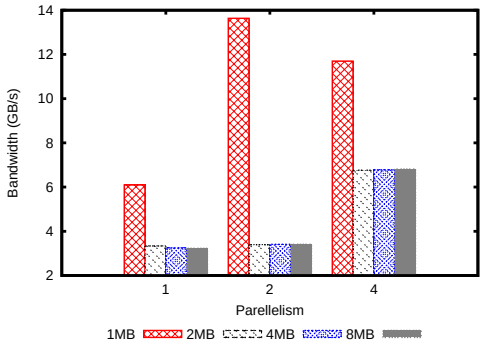
Figure 4.5: Memory Read-Write Bandwidth Results for Different Data Sizes



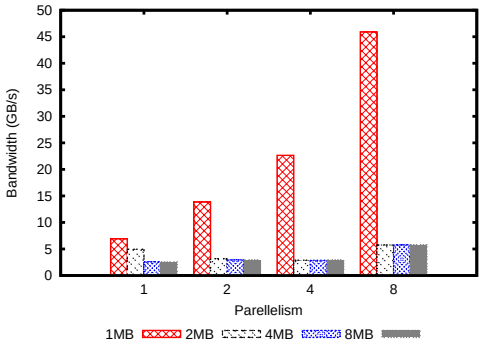
(a): Conroe



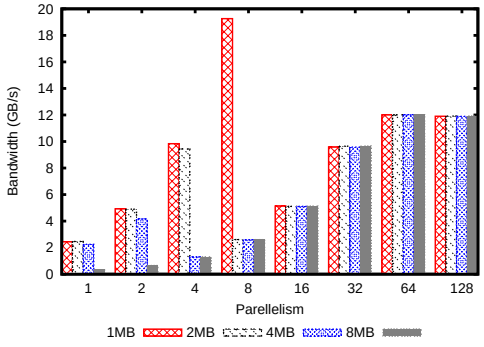
(b): Clovertown



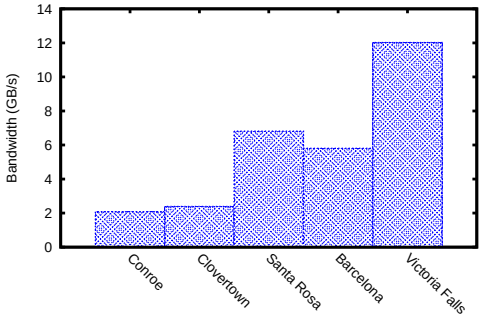
(c): Santa Rosa



(d): Barcelona



(e): Victoria Falls



(f): Peak Results (8MB)

Figure 4.6: Memory Write Bandwidth Results for Different Data Sizes

consistent performance behavior. It is important to note that Santa Rosa testbed has a 13% higher clock rate than the Barcelona testbed that explains the peak values difference between the two CMPs. However, the results reveal the fact that Barcelona is not getting any extra performance with the additional cores compared to Santa Rosa for this test.

Memory Read Bandwidth results are plotted in Figure 4.4. Victoria Falls is showing good scalability again with a 30 GB/s peak performance at 128 threads. In order to better interpret these results, it should be noted that Santa Rosa and Barcelona architectures follow the NUMA memory model, whereas the rest of the architectures represent a UMA model. Having twice the number of cores compared to Santa Rosa, in this test Barcelona performs the second best and outperforms Santa Rosa unlike the memory copy bandwidth test. Both Conroe and Clovertown is performing poorly in this benchmark compared to the rest of the testbeds. Figure 4.5 presents the results from the Memory Read-Write benchmark. This test measures the time to read data into memory and then write data to the same memory location. Victoria Falls has again the better scaling and peak value performance. Conroe and Clovertown shows good performance for 1MB and 2MB tests by benefiting from the larger cache size in these architectures. Santa Rosa has the second best scaling and peak value performance. Barcelona does not provide the extra performance expected compared to similar Opteron architecture Santa Rosa by having more cores. This is another illustration of the fact that performance scaling inside the multi-core architectures with the additional cores is hard to achieve during these stress tests. Figure 4.6 presents experimental results from the Memory Write bandwidth benchmark. Results from each of the testbeds are provided as well as comparison of the peak results. Same peak performance trend continues with Victoria Falls being the best and the Santa Rosa being the second best performing architecture.

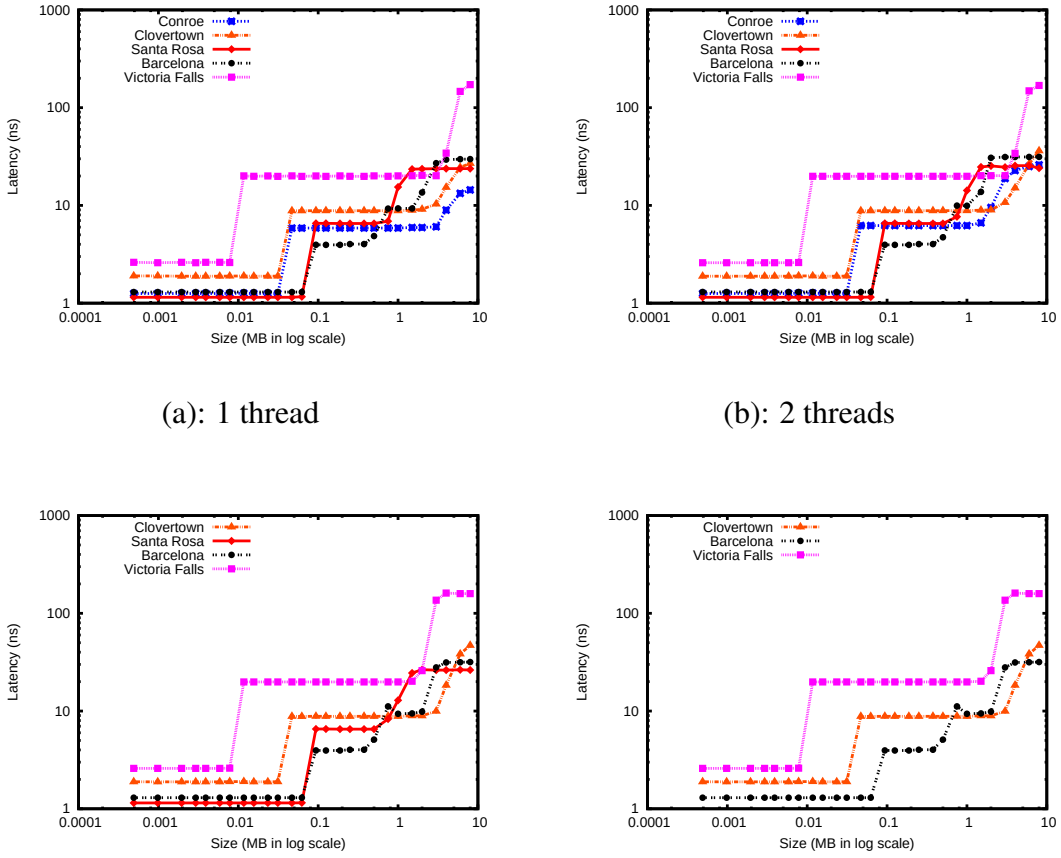
These memory bandwidth results again show that simpler but efficient multi-threading hardwares can outperform typical CMPs that combine multiple cores with a network con-

nection (e.g. a shared bus). Most of the scalability problems are observed again when two or more cores are competing for a shared resource such as the memory controller or the shared bus. In these scenarios, results demonstrate the fact that NUMA architectures can outperform its UMA/SMP counterparts and show better scalability by having separate paths to those shared resources.

In addition to the memory bandwidth analysis, some tests were conducted to evaluate memory latency performance. That analysis can be found in the next subsection.

4.1.2 Memory Latency Analysis

In this study, memory read latency is measured for varying memory sizes up to 8MB for 64-bit stride memory accesses. The entire memory hierarchy is measured starting from the L1 cache to the main memory. Only data accesses are measured discarding the instruction cache. Figure 4.7 presents the corresponding results from the Memory Read Latency benchmark from LMbench. The x-axis shows the size of the memory transactions in log scale whereas the y-axis represents the latency in nanoseconds (ns). In order to present the performance with different number of threads trying to access the data simultaneously, results for different thread counts are plotted for 1, 2, 4, and 8 threads. Santa Rosa and Barcelona show the best results in general, where Conroe performs very close and even better for a single thread test at the memory level. Santa Rosa and Barcelona are examples of a NUMA memory model. Also, they have private L2 caches that help them to achieve better performance in this study. This shows the importance of utilizing private caches for improved latency and better scalability. Accordingly, we also experimented a tiled-CMP architecture with private caches. Victoria Falls is performing relatively bad, but it is important to note that Victoria Falls performance benefits will emerge with the increasing number of threads as it is a multi-threaded architecture. All of the testbeds but Barcelona have two



(a): 1 thread

(b): 2 threads

(c): 4 threads

(d): 8 threads

Figure 4.7: Memory Read Latency Results

levels of caches. Utilization of a three-level cache hierarchy inside Barcelona processor shows better latency results. As such, these results also illustrate the importance of having multiple-levels of hierarchy in the memory system including multiple-levels of caches to alleviate the burden of high off-chip memory latencies.

4.1.3 Cache Coherence Overhead Analysis

We developed a synthetic benchmark in order to estimate the timing effects of cache coherence overhead as a part of our earlier work [95], which focused on identifying the per-

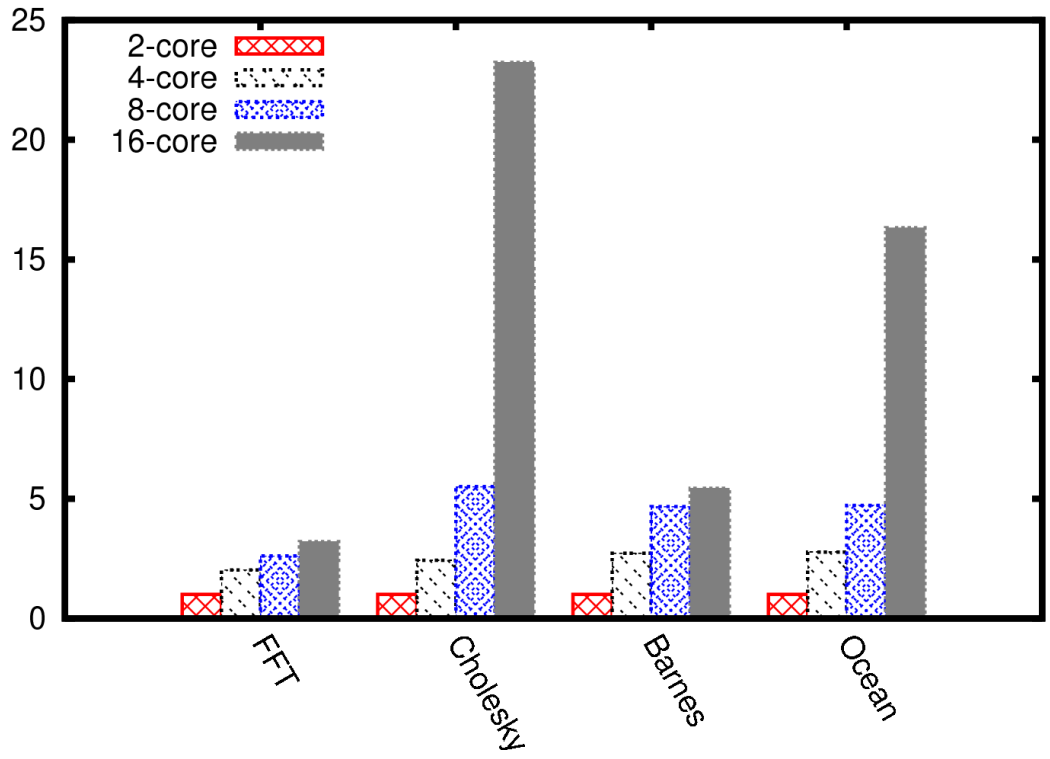


Figure 4.8: Normalized Cache Coherence Misses with respect to 2-core machine results

formance bottlenecks in CMPs. However, it is very hard to eliminate the system level overhead, mainly the operating system effect from the timing results. Thus, we decided to go with a simulation study in which various protocol messages and state transitions can be isolated to get an accurate measurement of coherence misses which can be used as a better metric to investigate cache coherence overhead. Also, the simulation study gives us the opportunity to investigate currently non-existing hardware. A full-system simulation was conducted to get more realistic results including operating system effects. Details of our simulation methodology was given in Section 3.3.

Four different target CMPs, with different number of cores were tested. The targets CMPs include 2-core, 4-core, 8-core, and 16-core SPARC CMPs. A snooping, broadcast based MOSI protocol is used in the simulated target systems. We focused on the coherence misses, especially how they scale for the same application with the increasing number of

cores. Corresponding results are presented in Figure 4.8, from four different target systems. The per core characteristics of all the target systems are identical as can be seen in Table 3.3, and the only difference between these different machines is the number of cores inside them. Simulation workloads include benchmarks from the SPLASH-2 benchmarking suite [117]; Barnes, FFT, Ocean, and Cholesky.

All the results are normalized to the coherence misses from a 2-core CMP for each benchmark. Coherence misses are calculated from the detailed cache state transition statistics provided by the GEMS simulation toolset [125]. All benchmarks exhibit increasing coherence misses with the increasing number of cores as expected, but Cholesky and Ocean show exponential increase on the 8-core and 16-core CMPs. This result shows that the cache coherence signaling overhead can become a performance bottleneck with the increased size of many-core architectures. In order to alleviate this, we propose to utilize adaptive cache coherence protocols that are optimized to reduce coherence misses. Furthermore, next section will show how the cache coherence protocol mechanism can affect the overall system performance.

4.2 Analyzing the Performance Impact of Cache Coherence in CMP-based systems

In this section, we present our performance characterization study, which focuses on the performance effects of high cache-to-cache latencies arising from cache coherence mechanisms. In doing so, we utilized relatively larger CMP-based systems that can reveal the performance issues that emerge from the underlying cache coherence protocol. This study was conducted on a 2312 compute-cores Sun cluster, Midnight Sun, based on Sun Fire X2200/X4600 servers at the Arctic Region Supercomputing Center (ARSC). As the ma-

chine uses two different compute node servers, the performance behavior of the system was characterized particularly in relation to the node level memory hierarchy performance using the two different node architectures based on the number of sockets: thin nodes and thick nodes. Thin nodes represent the two socket Sun Fire X2200 M2 servers and thick nodes refer to the eight socket Sun Fire X4600 servers. Corresponding node architectures were depicted in Figure 3.4. More details about the comprehensive study can be found in our related publications [22, 121, 124]. We start with some background information on the cache coherence protocol used in this testbed, as it plays an important role in the following discussions and analyses.

4.2.1 AMD Cache Coherence Protocol

The cache coherence protocol utilized by the AMD Opteron architecture is a snooping protocol with the MOESI coherence states [18]. This protocol adds a level of indirection to establish coherence ordering on an unordered interconnect namely HyperTransport [123]. The memory controllers, as seen in Figure 3.4, serve as the ordering point for cache coherence. As such, any coherence broadcast message from a core is first sent to its memory controller, *ordering point*, to establish the total order. Then the memory controller broadcasts the request message to all other processors in the system. The ordering point also blocks the current cache line address to prevent any race conditions [21].

The requesting core must wait explicit acknowledgement (ACK) messages from the other cores in the system to ensure the completion of a load or store operation. After the requester receives all acknowledgements and data, the memory controller unblocks the corresponding cache line [21]. This sequence continues for each request. This traffic was found to be a burden on the architecture and a source of performance imbalance among the cores depending on their placement within the node topology in asymmetric networks. In addition,

this broadcast signal traffic affects the thick nodes more severely than the thin nodes due to average longer travel times. This means as the number of cores grow and the network average distance increases, performance will also degrade. Detailed experimental results and analyses will be given in the following sections which lead to this conclusion.

4.2.2 Experimental Setup Overview

All the results were obtained under Suse Enterprise Linux 9.3 with kernel 2.6.5-7.276-smp. In addition, the gcc v3 compilers and PathScale MPI compilers were used for experiments. To get an overall understanding of the system level performance, experimental analyses began with real-world scientific applications described earlier in Section 3.2.2.

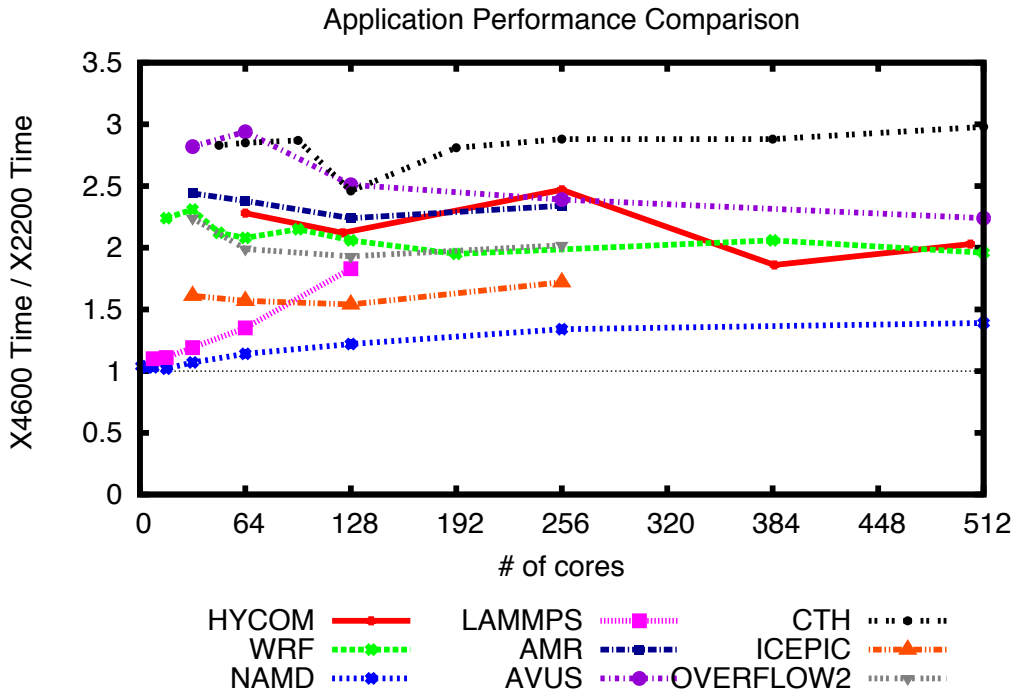


Figure 4.9: Application performance comparison between thin nodes and thick nodes

Figure 4.9 presents an overview of the performance of the scientific applications comparing thin and thick node performance. The x-axis in the figure shows the number of processors used in the tests and the y-axis shows the total execution time on the thick nodes divided by the total execution time on the thin nodes. It is clearly seen that the thin nodes are uniformly outperforming the thick nodes for all processor counts on these applications.

In order to further examine the performance issues, various benchmarks were used. From the HPC Challenge Benchmarks [98], the bandwidth effective (b_eff) benchmark [103] was used for the effective bandwidth investigations both at the intra-node and the inter-node level. The FFT benchmark was used to analyze the scaling at the system level since this kernel exhibits a balanced ratio of computation and communication. The STREAM benchmark [101] was used to obtain the sustained bandwidth and also to examine the performance balance among different cores inside the system. It is also modified to investigate the overhead induced by the cache coherence mechanism. In order to examine memory latency, LMbench v3 [99] was used. Also, some low-level micro benchmarks were developed for testing the memory copy performance to different memory regions. These benchmarks investigate potential performance bottlenecks and characterize the performance impact of cache coherence overhead with respect to these bottlenecks.

In the following section, more details of the experimental analysis will be given and observations will be provided. The selected benchmarks are divided into the following categories: Performance Scaling Analysis, Memory Bandwidth Analysis, Memory Latency Analysis, and Operating System (OS) Related Analysis. Even though some of the benchmarks fall into more than one category, the results from each benchmark will be examined within their most related category. In fact, all these different analyses include some level of observations for the cache coherence overhead analysis. In addition, the experiments are discussed starting at the system level and proceeding down to the core level in order to maintain a coherent flow to the discussion.

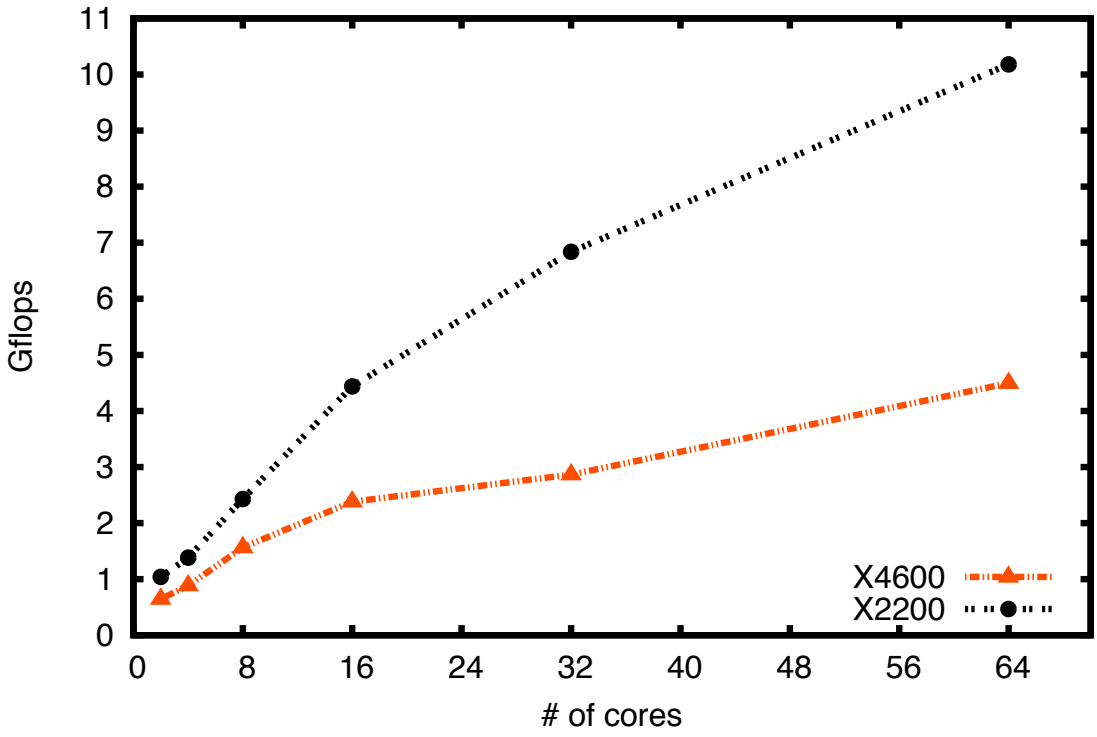


Figure 4.10: FFT Results (thin nodes vs. thick nodes)

4.2.3 Performance Scaling Analysis

HPC Challenge benchmark test, FFT, was used to analyze the performance scaling comparison of the two different compute nodes inside the cluster. FFT measures the floating point rate of execution of a double precision complex one-dimensional Discrete Fourier Transform (DFT) [98].

Figure 4.10 illustrates the FFT results that can be analyzed by focusing on two different regions of the performance curves. First, the results from two to sixteen processors compare the intra-node scaling of thick nodes with the inter-node scaling of thin nodes. As such, in this region the runs stay inside the thick node but need to span four thin nodes. It is shown that even the inter-node scaling of thin nodes is better than the intra-node scaling of thick nodes. This is even though the HyperTransport (HT) bandwidth is higher than the Infiniband (IB) bandwidth. Second, the results from 16 to 64 processors show that the same trend

continues with the thick nodes beginning to perform a little worse than the two to sixteen processors region, as they start facing the inter-node bandwidth after sixteen processors. Results indicate potential contention issues for thick nodes, which result in performance loss compared to the thin nodes. To isolate the performance bottlenecks causing this performance mismatch low-level intra-node micro-benchmarks are utilized and they will be discussed in the following sections.

4.2.4 Memory Bandwidth Analysis

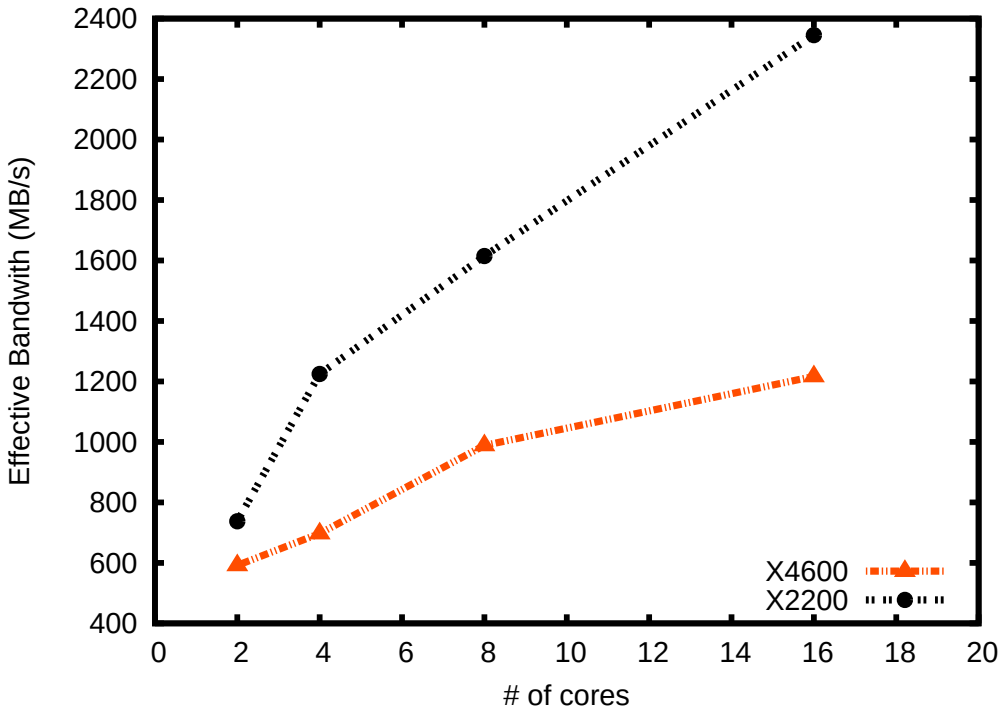


Figure 4.11: Effective Bandwidth Results (thin nodes vs. thick nodes)

For memory bandwidth experiments, two other kernels from the HPC Challenge benchmark, effective bandwidth benchmark (b_eff) and STREAM benchmark were used. The effective bandwidth benchmark, b_eff, measures the logarithmic average of accumulated

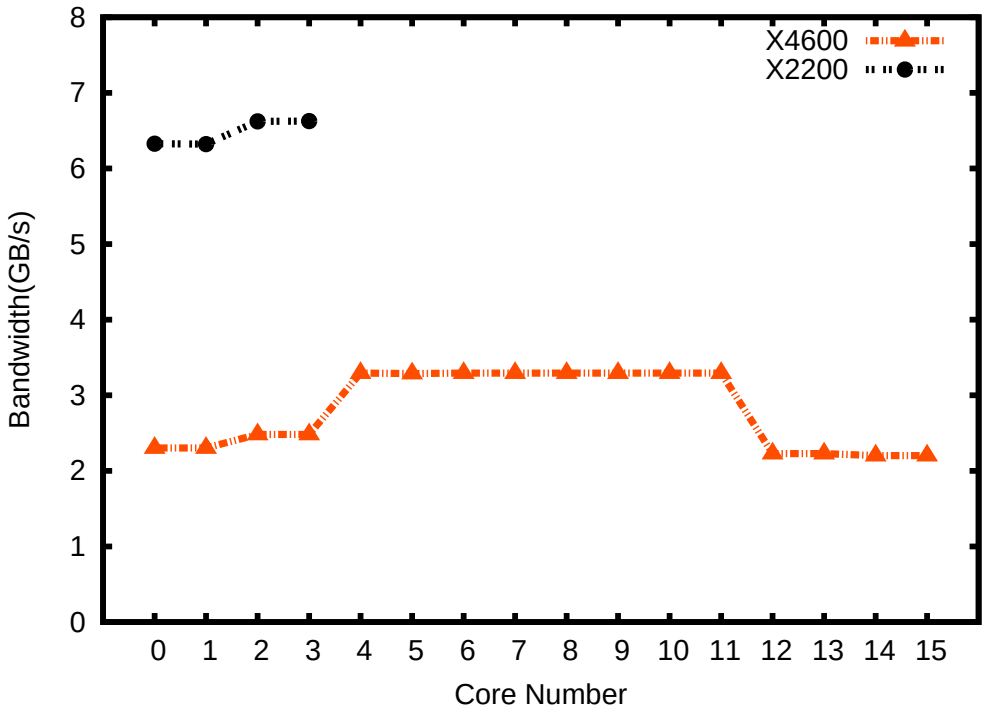


Figure 4.12: STREAM TRIAD results by cpu binding (array size = 1,5M)

bandwidth results of various message sizes of the communication network of parallel systems.

Figure 4.11 presents results between two to sixteen processors so that the runs for a thick node remained inside a single node whereas four nodes were used to reach this processor count by going through the Infiniband network. The theoretical uni-directional inter-node (Infiniband) peak bandwidth is 1GB/sec whereas the intra-node (HyperTransport) uni-directional peak bandwidth is 4GB/sec. As such, the intra-node bandwidth results from both of the systems are better than the inter-node bandwidth values as can be seen in Figure 4.10 and Figure 4.11. Accordingly, it is reasonable to expect the thick nodes to perform better in this effective bandwidth benchmark at least up to sixteen processors which were exactly the data points analyzed in Figure 4.11. Yet, this is not the case as can be ob-

served by looking at the slope of both plots and final values. The results are following the same trend as the FFT results that were illustrated in Figure 4.10. It can be concluded that the contention inside the thick nodes severely affects the performance so that these nodes cannot exploit the higher intra-node bandwidth. Therefore, following experiments will focus more on the intra-node performance of thick nodes to identify the contention issues which were shown to have a big impact on the overall system performance. It should be noted, however, that this behavior could be because of the higher cost of cache coherence mechanism in the thick nodes, which will be investigated later.

The STREAM benchmark is also part of the HPC Challenge Benchmarks [98] and is intended to test the sustained memory bandwidth of a system. However, we modified this test by reducing the array size to stay in cache and to examine the cache coherence overhead. Also, the STREAM benchmark was utilized to test the performance balance among the processors inside the nodes. We did our STREAM tests for each and every core in the nodes by binding the process to those cores using the "taskset" command. Taskset is a command line utility in Linux for binding the processes to desired cores [126].

Figure 4.12 presents interesting results from the STREAM TRIAD benchmark as we see a major performance mismatch between various cores of the thick node system. There are a couple of observations that can be reached by examining Figure 4.12. First, the performance mismatch between the two nodes types is still present. However, this experiment tests sequential runs and presents single thread results. Accordingly, the results should follow a flat line for each node type and the two lines for each node should be very close since the core architectures are almost identical. Thus, it is also observed that performance mismatch goes down to core level. Finally, if we refer to cores 0-3 and 12-15 as cores on the edges and cores 4-11 as middle cores; it is shown that middle cores are getting a lot more bandwidth than the cores on the edges. To further examine this performance difference, we utilized LMbench [99] to investigate memory latency performance. The corresponding

results will be investigated in the next section.

4.2.5 Memory Latency Analysis

As it is shown in Figure 4.12, the middle cores are outperforming the cores on the edges. Accordingly, the following hypotheses were devised to explain this performance mismatch:

- Hypothesis 1: There is an extra overhead for cores on the edges compared to the middle cores.
- Hypothesis 2: There is a problem in enforcing memory affinity.

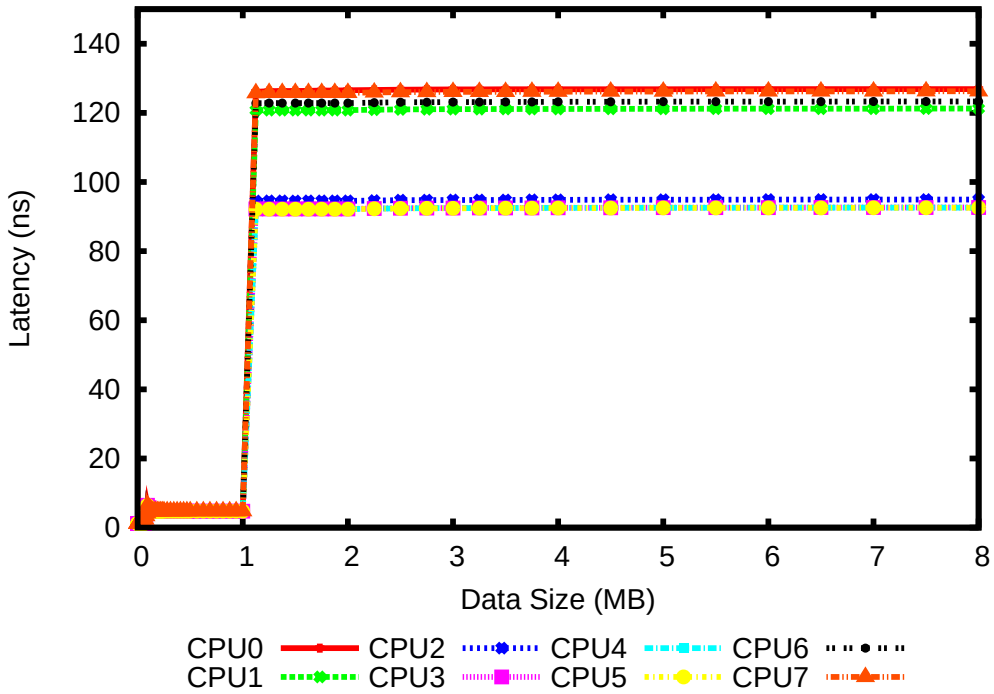


Figure 4.13: Memory Read Latency results from a thick node

In order to get a better understanding for the first hypothesis, we utilized LMbench [99]. The memory read latency results were measured for all levels of memory hierarchy inside the cores using LMbench. Figure 4.13 represents the corresponding results from a thick node. This result actually indicates that the memory latency from all the cores are almost the same until the capacity limit of the L2 caches (1MB) is reached, and after that a major performance deviation is observed between the middle cores and cores on the edges. This behavior can be explained by the snooping protocol mechanism adapted by AMD64 platforms. At the time of a cache miss, which is the point where this deviation starts to occur, probe signals are sent to other processors to check whether they have the recent copy of the requested data in their cache lines even before checking the local memory. This traffic results in a greater overhead for the cores of the CMPs on the edges compared to the middle CMPs with respect to the node topology that was shown in Figure 3.4.

In order to investigate the second hypothesis which basically questions the memory affinity effect in the nodes, some experiments were conducted on the operating system level which will be explained in the next section.

4.2.6 Operating System (OS) Related Analysis

In order to investigate the second hypothesis, the numastat statistics counter, provided by the Linux NUMA API was utilized [126]. Numastat provides information about the outcomes of NUMA allocation calls, such as information about whether the memory is successfully allocated in the local memory or not, and can be obtained by looking at the results for $numa_hit$ and $numa_miss$ values. This counter accumulates statistical values reported from the Linux kernel. Thus, the *numastat* command was executed before and after the memory copy micro-benchmark (64 MB) to get accurate information about the NUMA allocation behavior.

Table 4.1: Numastat Output Before and After the Memory Copy Benchmark From Socket #7

Counter name	Before	After
$numa_hit$	34869658	34874651
$numa_miss$	***3390900***	***3390900***

The results presented in Table 4.1 show that memory allocation was successfully accomplished inside the targeted processors local node since the $numa_hit$ value increases whereas the $numa_miss$ value remains the same. This finding eliminates the memory locality issue as a reason for the performance mismatch that was observed inside a thick node and leaves hypothesis #1 as the major reason for the performance mismatch inside thick nodes.

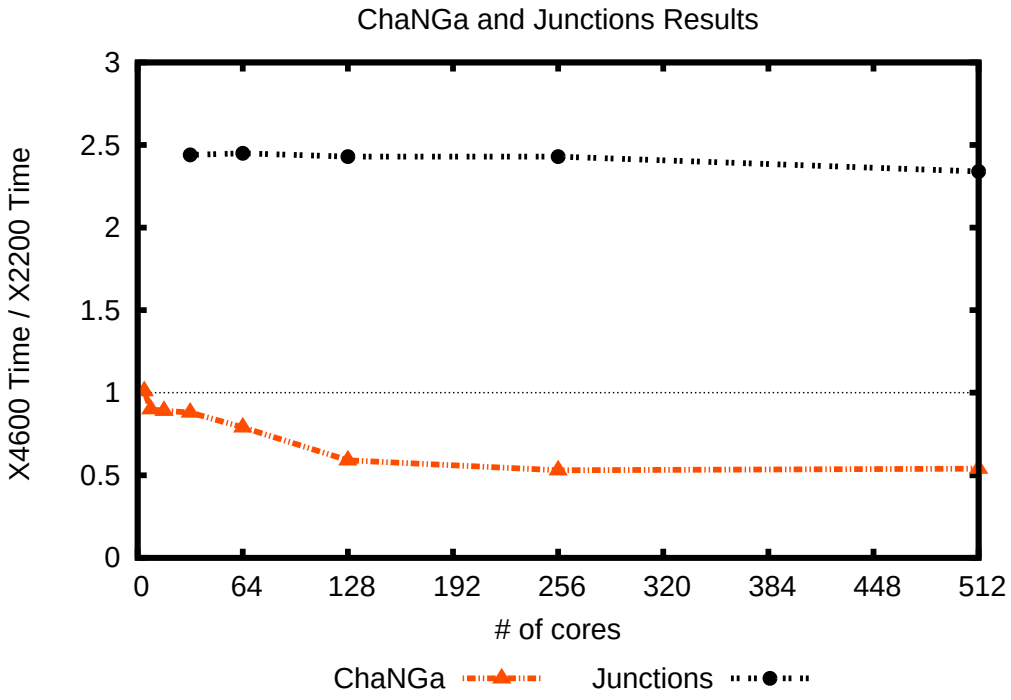


Figure 4.14: Application Performance Comparison Junctions vs. ChaNGa

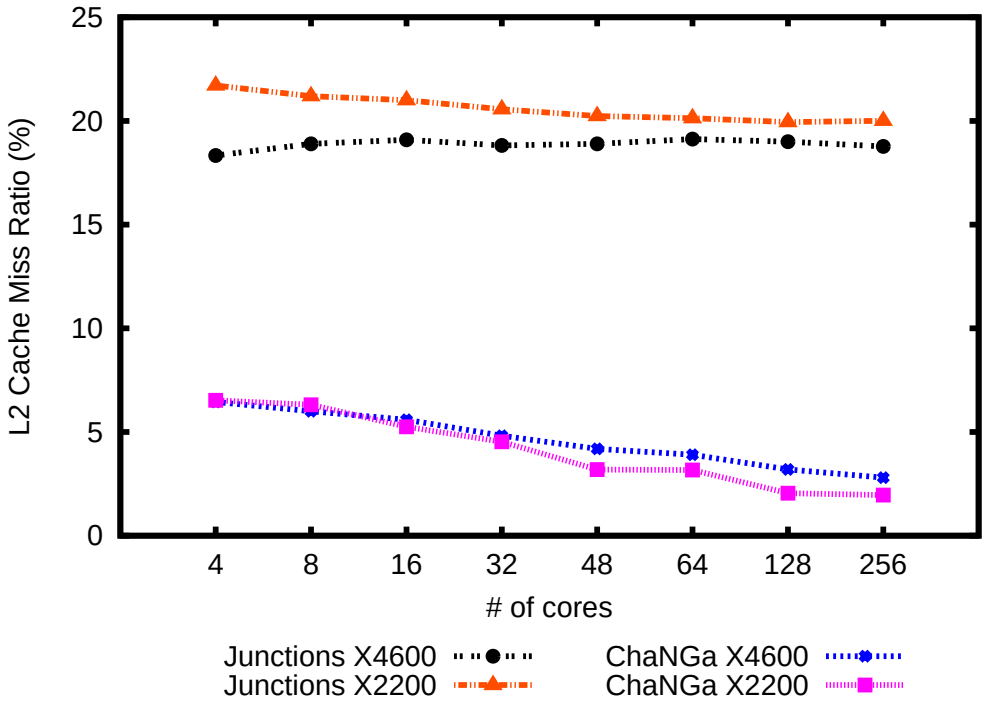


Figure 4.15: L2 Cache Statistics of Junctions and ChaNGa

4.2.7 Performance Analysis Using Hardware Counters

To further investigate the cache coherence overhead and its effect on the observed performance mismatch, we did some experiments using performance counters. In order to obtain hardware counter statistics, the Performance Application Programming Interface (PAPI) [100] was utilized.

Figure 4.14 shows an application performance comparison between two applications, Junctions and ChaNGa, similar to Figure 4.9. Junctions is another scientific application, a solver based on LAPACK and BLAS, for finding properties of materials. As this figure shows, Junctions has a similar performance pattern to the other codes with the thin nodes outperforming the thick nodes, whereas for ChaNGa the thick nodes are outperforming the thin

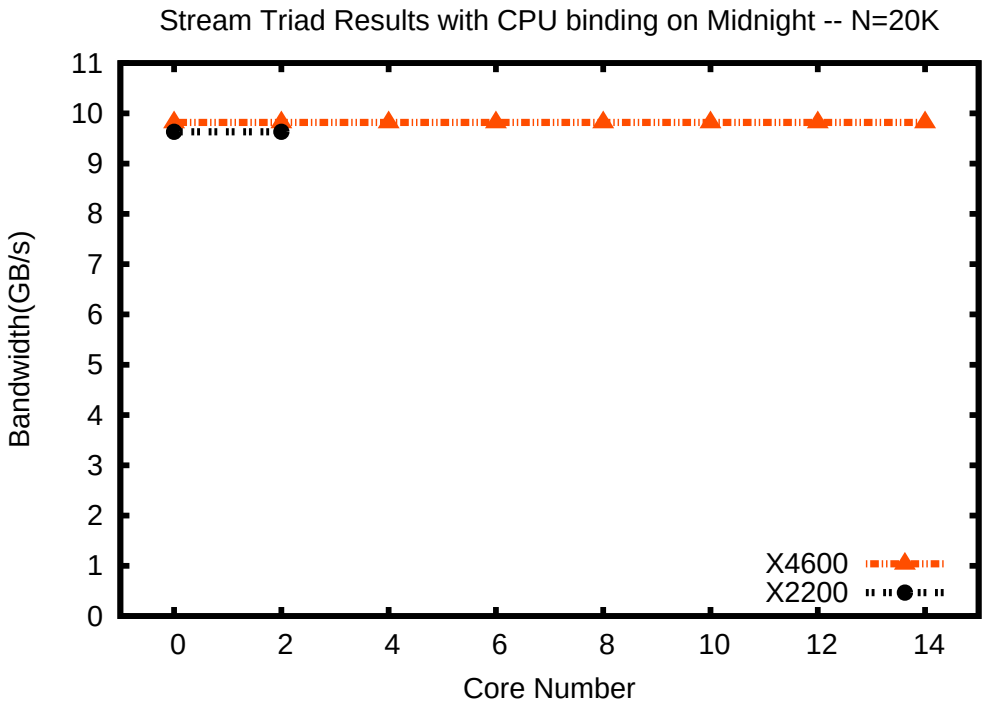


Figure 4.16: STREAM TRIAD results by cpu binding (Array Size = 20K)

nodes. Out of all the applications and benchmarks tested in this work, only ChaNGa performed in favor of thick nodes. Thus, analysis of ChaNGa becomes crucial to verify our observations and findings.

In order to examine the behavior in Figure 4.14, experiments were repeated using PAPI to get L2 cache statistics for these two applications. Figure 4.15 illustrates the corresponding results obtained from the hardware performance counters. The y-axis represents the L2 Cache Miss ratio. For ChaNGa, this value is relatively small for a scientific application and is less than 5% in most of the core counts. On the other hand, Junctions has around 20% L2 Cache Miss ratio. It can be concluded that ChaNGa is more cache-friendly than Junctions. This also results in fewer cache coherence broadcast messages, which supports our findings on the impact of cache coherence overhead to performance mismatch.

In addition, the STREAM TRIAD benchmark test was also repeated using PAPI to get L2 cache miss statistics. Since the STREAM benchmark is designed to avoid the L2 cache and instead fetch data only from main memory, the original STREAM run resulted in almost 99% L2 cache misses. To be able to obtain a cache-friendly version, the array size was reduced to $20K$ elements to stay in L2 cache for the calculations. This modified version of STREAM had around 2% L2 cache misses. The corresponding bandwidth results can be seen in Figure 4.16. Compared to the previous STREAM test that was plotted in Figure 4.12, now we see a flat line for both node architectures and that the node types are performing very close to each other. This leads to the conclusion that the major contention comes from maintaining the cache coherence inside the nodes. Since thick nodes have a greater number of sockets, this contention affects the thick nodes more severely than the thin nodes. Therefore, an increase in cache utilization not only improves performance in general but also results in better performance for thick nodes compared to thin nodes, since bandwidth and latency will favor thick nodes in general.

4.3 Summary

According to the experimental results, it is concluded that the major overhead inside these cluster nodes is due to cache coherence overhead. The reason that cores on the edges suffer more from this overhead is the fact that cache coherence snooping traffic overhead is a direct reflection of the architecture that we were studying. In a thick node for example, cpu0 is always bound to the three-hop broadcast signal latency that it needs to receive all responses from every processor inside the node; likewise for cpu7 which is also bound to a latency of three hops that will be coming from cpu0. Consequently, these cores were found to misbehave and suffer severely from this architectural feature. This also explains the overall system performance mismatch as the thick nodes will be more severely affected

by this cache coherence signaling traffic. On the other hand, ChaNGa, which is written in CHARM++ and employs intelligent dynamic load balancing mechanisms, performed better on the larger nodes. ChaNGa seems to achieve that by balancing the load and intelligently improving locality within the nodes to a point that made the per-core-performance for the larger node close to that of the smaller node. Once this was achieved, additional benefits were gained from the shorter average communication distance at the system level when larger nodes are utilized, resulting in overall better performance than that of the smaller nodes.

It is therefore noted that the cache coherence protocol performance can have severe affects for a wide range of platforms. It is also shown that the affects become even larger with the increasing number of cores. Accordingly, future CMPs must deploy efficient cache coherence mechanisms to be able to harness the potential performance benefits. As such, there is a need for design and exploration of cache coherence protocols that are targeting CMPs. Furthermore, it is demonstrated that symmetry of network on-chip is important, and can become an issue unless intelligently designed. As such, upcoming chapters provide the details of our proposed cache coherence schemes that are designed according to these observations and can adaptively work to improve the performance in CMP based systems.

Chapter 5

Producer-Consumer Adaptive Cache Coherence Protocols for CMPs

This chapter describes our work on cache coherence optimizations for CMPs. In doing so, first of all, the design considerations are discussed including the target CMP architecture and the baseline directory-based cache coherence protocol. Then, starting with the Producer-Consumer Adaptive Multicore (PCAM) protocol, details about the proposed cache coherence protocols are given in this chapter. The target CMP architecture as well as the proposed cache coherence mechanisms are directly motivated by the findings of the previous chapter. At the end, we also provide the details of the analytical model.

5.1 Design Considerations

Thread-level parallelism (TLP) should be exploited to harness the potential performance from CMPs. Shared-memory programming model provides an easier solution to exploit TLP. Accordingly, most CMP systems have adapted the shared memory model similar to

the earlier MPs (traditional SMP and ccNUMA systems). Shared memory model entails communication primitives such as cache coherence and synchronization mechanisms, in order to maintain correctness while concurrently running multiple threads. Such mechanisms in CMPs can benefit from the earlier research studies for conventional MPs. However, treating CMP based systems as conventional MPs would miss significant new opportunities as CMPs offer peculiar capabilities that are fundamentally different from traditional MPs [25]. Some of these differences as stated earlier in Section 1.2, can be summarized as follows:

- Latencies between cores in CMPs are far less compared to typical SMPs.
- The relative latency difference between nodes was not significant in typical MPs but it is now in CMPs.
- From any core/node in the system, memory is no longer at a comparable distance as the neighboring cores/nodes. Off-chip memory access latency is much higher than on-chip cache-to-cache access latency.
- The inter-core bandwidth on a CMP can be much higher compared to a traditional SMP.

According to the International Technology Roadmap for Semiconductors (ITRS), the off-chip bandwidth is expected to increase linearly rather than exponentially. Thus, a high on-chip cache performance is imperative to mitigate the off-chip bandwidth bottleneck [23]. All these differences make it important to revisit the design of scalable cache coherence protocols for CMPs in this new context [23].

Contemporary architectures employ write-invalidate based cache coherence protocols that are known to generate coherence misses that later produce additional latency. This is especially true under producer-consumer sharing patterns and can become a performance

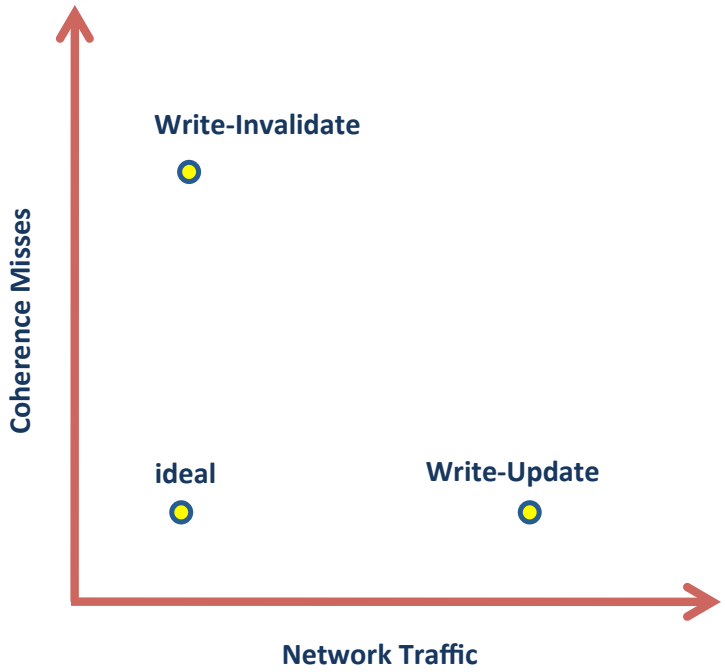


Figure 5.1: Write-Invalidate vs. Write-Update protocol in terms of network traffic and coherence misses

bottleneck as the number of cores increases. On the other hand, write-update based cache coherence protocols can solve the coherence misses problem, but they tend to generate excessive network traffic. Figure 5.1 demonstrates this by roughly comparing write-invalidate based protocols and write-update based protocols in terms of network traffic and coherence misses. Accordingly, previous studies have shown that a single protocol approach is not sufficient for many sharing patterns. As mentioned earlier, in contrast to traditional multiprocessor systems, relative latency between cores will vary a lot as the number of cores increases within a CMP. Moreover, it may require dozens of cycles for a signal to traverse between two opposite edges of the die [21, 23]. Thereby, this thesis provides solutions to alleviate the burden of these remote misses under write-invalidate protocols that would become even more important in future CMPs.

Cache coherence schemes in general are tightly coupled to the ordering properties provided by the underlying interconnect [21]. Buses provide simpler interface with their total order-

ing properties but will suffer to scale after a moderate number of cores. As the number of cores increases inside a single CMP; low-power, high-bandwidth, scalable on-chip communication substrate is imperative to enable scaling to large core counts [26]. Furthermore, on-chip interconnects for future CMPs will have different design constraints with tighter resource budget compared to the earlier multiprocessors [21]. Therefore, our proposed protocols are based on scalable directory protocols that can work on unordered interconnects (e.g mesh, torus). In addition, this thesis also provides novel adaptive mechanisms that can dynamically adjust to varying interconnection resources.

5.1.1 Tiled-CMP Architecture

Tiled-CMP architectures provide a scalable and efficient solution for CMP design [11,128]. Such architectures are built by replicating the compute elements (tiles), which are usually comprised of simple cores and basic networking infrastructure. TILE64 [11] presents a commercially available example of such a tiled-CMP architecture. Details of the TILE64 platform can be seen in Figure 5.2. TILE64 includes a 2D grid of 64 cores/tiles. Each tile includes a processor and its cache hierarchy along with a switch that is used to connect to neighboring tiles [129]. Some other known examples of tiled architectures include RAW [130], SmartMemories [131], TRIPS [132], and Wavescalar [133].

Accordingly for the experiments, we utilized a tiled-CMP with 16 or more cores designed via 2D-MESH interconnect topology. Figure 5.3 depicts the architecture diagram of the experimented tiled-CMPs. Each tile contains an in-order, dual-issue UltraSPARC-III Cu processor with 2GHz clock frequency. The first level cache is a split-cache with 4-way set-associative 64KB instruction and data caches. Second level caches are private, unified 4-way set-associative 2MB caches. Each tile contains its own directory cache by holding only the blocks that are homed by the corresponding core. On-chip directory lookup la-

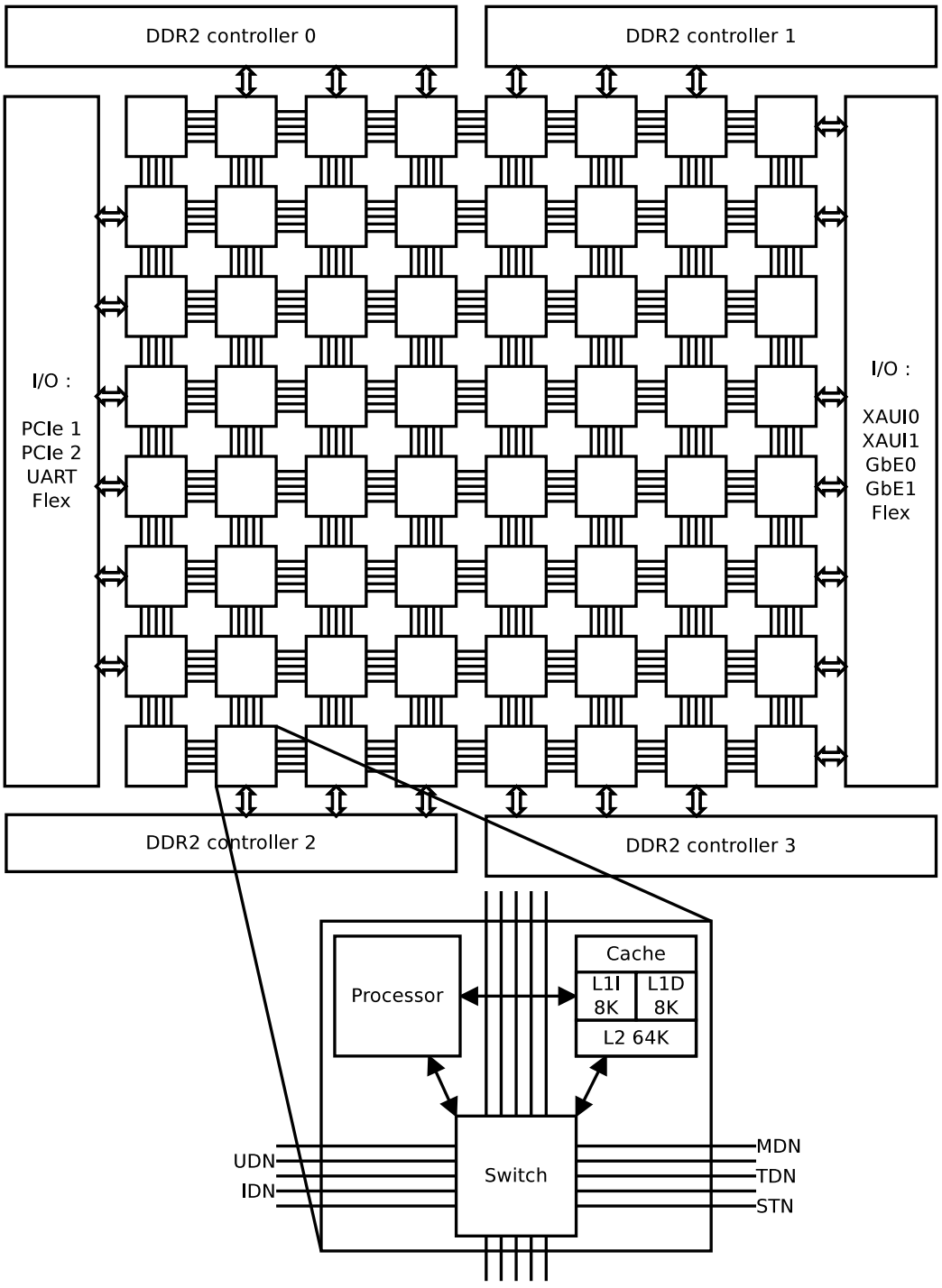


Figure 5.2: TILE64 Architecture Diagram [11, 127]

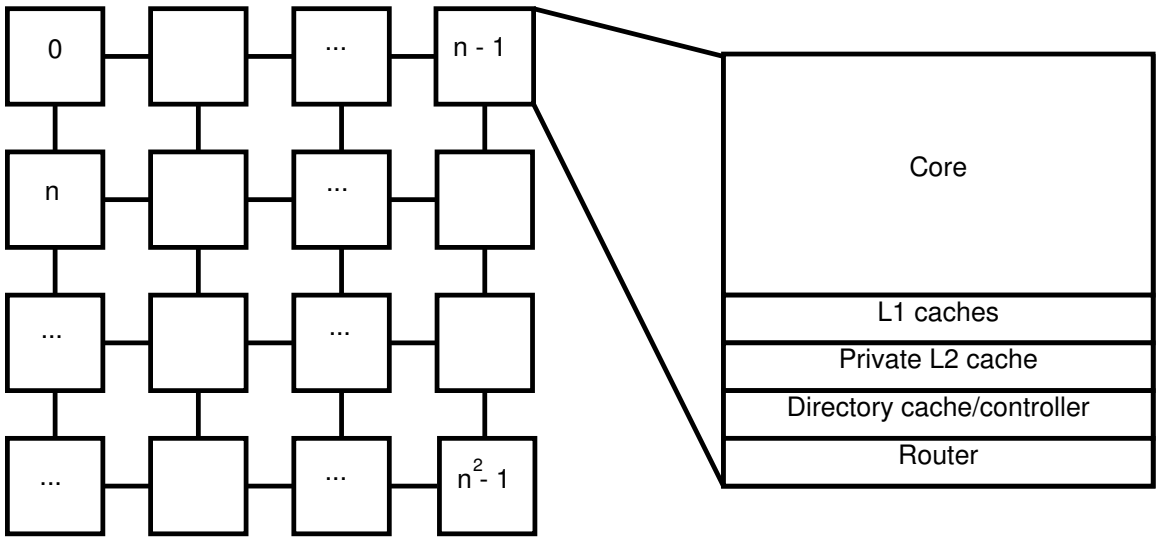


Figure 5.3: Target tiled-CMP architecture, connected via nxn 2D-MESH topology ($n = 4, 8, 16$)

Table 5.1: Simulation Architectural Details

Parameter	**Value**
Processors	In-order 2-way UltraSPARC-III Cu processor, 2GHz
# of cores	16, 64, 256
L1 Caches	Split I&D, 64KB 4-way set associative with 64-byte blocks per core
L2 Caches	4-way set associative with 64-byte blocks per core
Directory Cache (DC)	8192 entries
Cache Replacement Policy	Pseudo-LRU
Coherence Mechanism	Directory-based protocol based on MOESI states
Memory	512MB per core shared memory, 100-cycle access latency
Interconnect	4x4 2D MESH, 4-cycle latency per hop

tency is 20 cycles. Cores are connected with an unordered 2D-MESH with 4-cycle hop latency. If not otherwise specified, link bandwidth between tiles is assumed to be 8 bytes per cycle. Also, the network is assumed to operate at a frequency of one fourth of the CPU, due to power and energy constraints. Table 5.1 presents the details of the target machine. Although we selected a tiled architecture for our experimental study, proposed cache coherence protocols can work on any CMP with private caches at some level of the memory subsystem.

5.1.2 Baseline Cache Coherence Protocol

In this thesis, all of the proposed adaptive, hybrid protocols are based on a write-invalidate directory protocol with MOESI states. This baseline protocol is a slightly modified version of the directory protocol that is a part of the GEMS distribution [125]. We removed the dirty-locally modified state (MM) which is used to invalidate the producer instead of putting it in Owned (O) state for speculative migratory sharing optimization. Since our adaptive/hybrid protocol implementations are based on this protocol, this modification does not provide any bias towards our adaptive protocols, but helps to reduce the number of states and start with a standard MOESI protocol. The Exclusive-clean (E) state obviates upgrade misses to non-shared data. Race conditions are resolved at the home node by using a busy/active state for each request. Order of arrivals at the home node establishes the order in which the racing requests are serviced [38].

Directory induced latency is a bottleneck in such a protocol. In addition, write-invalidate protocols tend to generate high cache-to-cache misses which can also further become a performance issue as the number of cores increases on a single-chip. Also, the same problem applies to multi-CMP systems. As such, the main goal of the proposed protocols is to mitigate the latency burden driven by the cache-to-cache misses. The following section describes the implementation details of our proposed protocols.

5.2 Key Concepts and Mechanisms

This section describes the details of the proposed cache coherence protocols. First, the necessary hardware extensions for the proposed protocols and the producer-consumer sharing pattern detection mechanism are described. Then, the bandwidth-adaptive mechanism and

the proximity-aware mechanism are illustrated. Furthermore, implementation details are given to integrate such mechanisms into the proposed cache coherence protocols. These proposed adaptive cache coherence protocols along with their abbreviations as referred to in this thesis are as follows:

- **PCAM:** Producer-Consumer Adaptive Multicore protocol
- **PCBAM:** Producer-Consumer Bandwidth Adaptive Multicore protocol
- **PCPAM:** Producer-Consumer Proximity Adaptive Multicore protocol

5.2.1 Producer-Consumer Predictor Cache

To achieve producer-consumer data sharing pattern detection, we extended the on-chip directory cache with some additional bits for each cache line. This directory cache extension is referred to as the "*Producer-Consumer Predictor Cache (PCPC)*". It uses the directory state information with some additional bits to track down the producer-consumer sharing status for a cache line. More information regarding the extended cache lines inside PCPC are given in the next section.

As can be seen in Figure 5.3, coherence directory tags are decoupled from the regular cache (L1 and L2) tags. This enables faster access to the directory information even the requested line is missing in the L2 of the home node. Producer-consumer sharing pattern prediction tracking is done in parallel with the cache accesses. PCPC is organized as a 4-way set-associative cache to exploit spatial locality. The cache lines are indexed by data block addresses with a size of 8K elements unless specified otherwise. Each core has its own PCPC tracking only the blocks homed by that node. The directory cache replacement policy is LRU. Although there are limitations with this approach, this study aims to analyze potential gains with such small hardware extensions as the area budget is crucial for CMPs.

5.2.2 Producer-Consumer Sharing Pattern Detection

Producer-Consumer sharing pattern can be defined via the following regular expression:

$$(W_a)(R_{\forall b:b\neq a})^m : a, b \in S,\ m \geq 1 \tag{5.1}$$

W_i and R_i refer to write and read operations by processor i. S refers to the complete list of processors in the system.

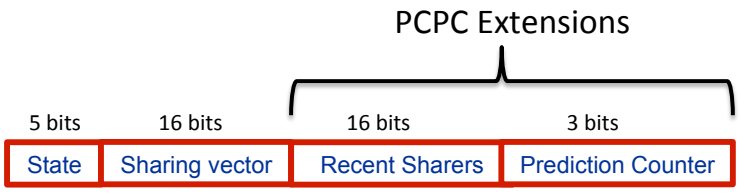


Figure 5.4: Extended Directory-Cache/PCPC Fields

As illustrated in Expression 5.1, producer-consumer sharing pattern is defined in which one processor modifies a data block (producer) and a set of processors read the block (consumers). To detect cache lines that exhibit producer-consumer sharing pattern, we extended each directory cache entry with a *recent_sharers* field to track the potential consumers as also suggested in [79]. In addition, a 3-bit saturating counter "*prediction_counter*" is used to track stable producer-consumer sharing pattern for a cache line. Although the optimal size of this prediction counter is application dependent, the 3-bit choice was experimentally shown to perform well for many of the applications tested as compared to 2 and 4 bits. For brevity and comparison purposes, all of our proposed protocols use this 3-bit saturating counter for the prediction mechanism. Figure 5.4 shows the directory cache entries extended with the *prediction_counter* and the *recent_sharers* fields. Since we experimented with a directory-cache with 8K entries, this resulted in just 19KB extra hardware cost per core to implement PCAM. The *recent_sharers* field is analogous to the sharing vector used

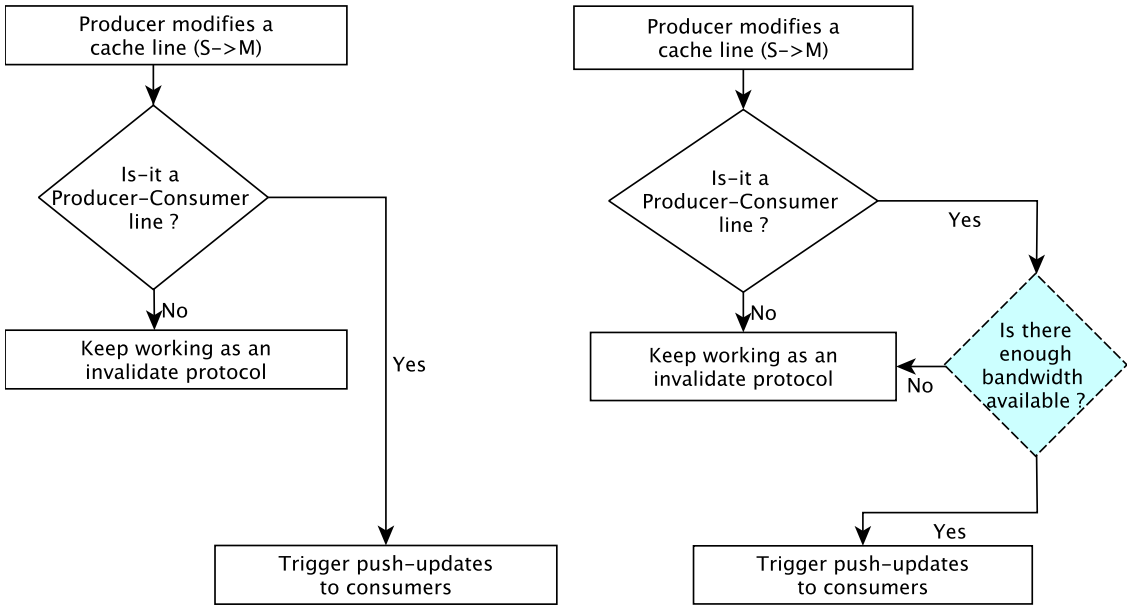
Algorithm 1 Algorithm to detect Producer-Consumer Sharing

```
1:  pred_cnt ← 0
2:  for all shared access to cache line A do
3:      if request is exclusive then
4:          sharers ← requester                          ▷ Sharing Vector is reset
5:      end if
6:      if request is shared then
7:          sharers ← sharers ∪ requester
8:      end if
9:      if request is shared or exclusive then
10:         if requester ∈ recent_sharers then
11:             pred_cnt ← min(pred_cnt + 1, 7)
12:         else
13:             recent_sharers ← recent_sharers ∪ requester
14:             pred_cnt ← max(pred_cnt − 1, 0)
15:         end if
16:     end if
17:     if pred_cnt = 7 then
18:         Mark cache line A as a producer/consumer
19:     end if
20: end for
21:             ▷ pred_cnt is referring to the Prediction Counter field in the directory cache
22:   ▷ pred_cnt and recent_sharers are reset when the line is evicted from the directory
    cache
```

by the directory and it represents a superset of the sharing vector with the addition of some other processors that recently accessed the same cache line. For each activity generated from the *recent_sharers* list, a 3-bit saturating *prediction_counter* is incremented. Also during each cache line replacement, the prediction history will be reset for that specific cache block. Tracking is only conducted for those lines cached by that specific core to minimize hardware requirements. Algorithm 1 demonstrates how we keep track of recent history of sharers to predict potential consumers in the future. To simplify the pseudo-code for clarity, operations on "*Sharing Vector*" and "*Recent Sharers*" fields are separated. As such, lines 2-8 show the operations on the "*Sharing Vector*" field and lines 9-15 show the operations on "*Recent Sharers*" field. Accordingly, for an exclusive access to a cache line, "*Sharing Vector*" is reset as in a typical directory (as shown in line 4). However, an exclusive access does not result in a reset for the "*Recent Sharers*" field (as shown in lines 9-15) in order

to keep a history of sharers for this cache line. When the *prediction_counter* is saturated, the corresponding cache line is marked as a producer/consumer line. Thereby, future modifications to this line generate updates to the potential consumers identified by the "*Recent Sharers*" field.

5.2.3 Bandwidth Adaptive Mechanism



(a): PCAM Protocol Flowchart (b): PCBAM Protocol Flowchart

Figure 5.5: Flowcharts depicting the producer-consumer adaptive and bandwidth-adaptive mechanisms inside proposed cache coherence protocols

Write-update based protocols tend to generate heavy network traffic as each modification to a shared cache line will cause data updates to all sharers. Having this is mind, our proposed protocols trigger updates only for those lines that are exhibiting a producer-consumer sharing pattern. Thereby, in our proposed protocols not only corresponding coherence misses are reduced but also the network traffic is kept low. However, there might be situations where either the network bandwidth is limited or a congestion is caused by a hot-spot in the network. Therefore, we propose a bandwidth-adaptive mechanism to alleviate the

burden of potential updates in such scenarios. This bandwidth-adaptive mechanism is implemented on top of the PCAM protocol resulting in the *Producer-Consumer Bandwidth Adaptive Multicore (PCBAM)* protocol.

A flowchart of how PCAM and PCBAM protocols are interrelated to each others and an overview of the underlying adaptive decision mechanisms can be seen in Figure 5.5. To summarize, two important adaptive decision mechanisms are required: producer-consumer (PC) detection mechanism and the bandwidth adaptive mechanism which is how PCBAM extends PCAM. PCBAM is designed to control network traffic generated by the write-updates in PCAM when the network has a limited bandwidth. PCBAM utilizes a bandwidth adaptive mechanism that is inspired from an earlier study by *Martin et al.* [14], *BASH*, which dynamically adapts to the available bandwidth to switch between a broadcast-based protocol to a directory like protocol. However, we utilize bandwidth-adaptive framework as augmented to the PCAM protocol to intelligently trigger write-update optimizations based on the available bandwidth. BASH uses a utilization counter in each core to calculate the link utilization and a policy counter to probabilistically decide whether to unicast or broadcast. Policy counter was mainly used to avoid oscillations in the system as BASH continuously needs to make decisions. However, our bandwidth adaptive policy comes only active for those cache blocks exhibiting producer-consumer sharing. Hence, we only implemented a signed utilization counter over a sampling interval of 256 cycles to identify if the link utilization is above or below a *cut-off threshold* (75%) as suggested in [14]. We also conducted a sensitivity analysis with varying static values between 25% and 95%, and it was concluded that 75% performs well in most cases as a threshold value. Experimental analysis which shows the importance of such a mechanism under high bandwidth utilization periods can be seen in the next chapter.

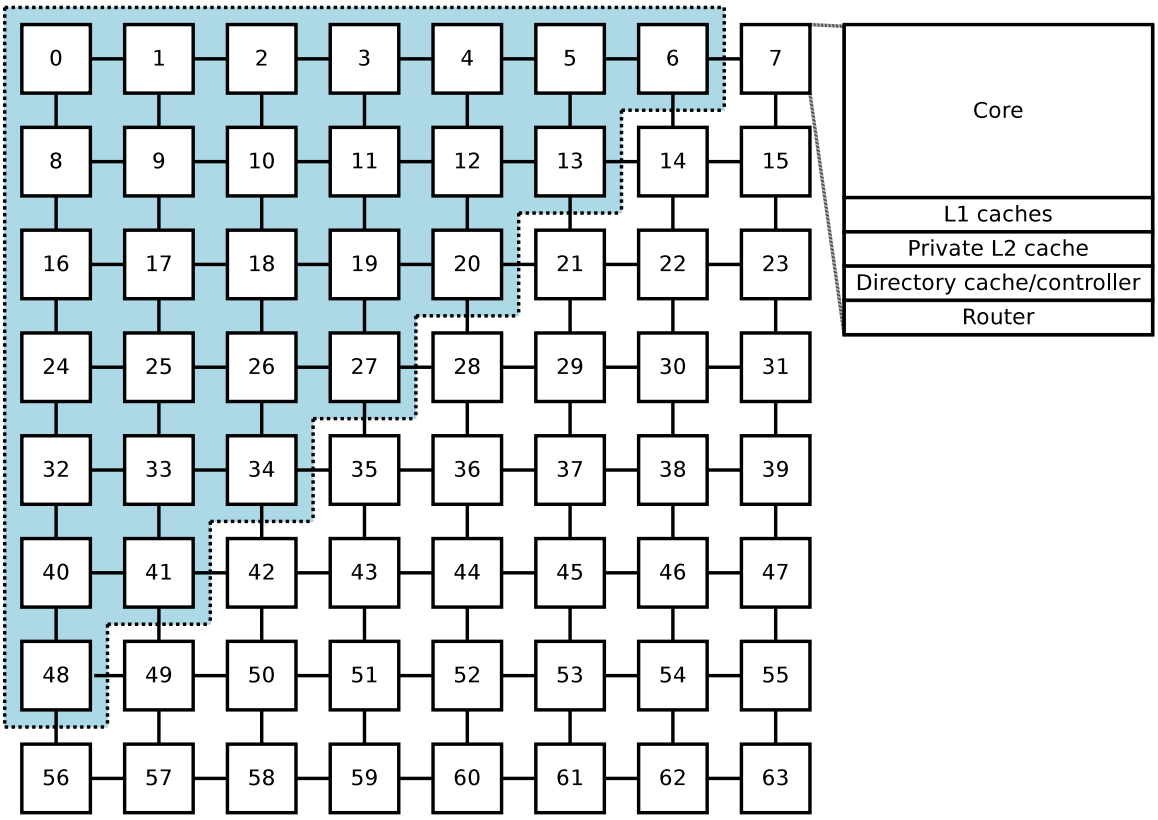


Figure 5.6: Target tiled-CMP architecture for PCPAM protocol study; a 64-core system connected via 8x8 2D-MESH topology

5.2.4 Proximity Aware Mechanism

Proximity-aware mechanism aims to establish latency based updates to consumers. It is based on the assumption that updates to far remote caches can be costly in terms of network traffic and accordingly may degrade performance. With this in mind, a proximity-aware mechanism is implemented on top of the PCAM protocol, and it is referred to as "*Producer-Consumer Proximity Adaptive Multicore (PCPAM)*" in this study. PCPAM pushes updates to consumers only within a certain proximity of the producer. This proximity is calculated utilizing the average link latency inside a 2-D Mesh interconnect topology which is shown in Equation 5.2.

$$avg.\ distance\ in\ d-dimensional\ k-ary\ mesh \approx d * \frac{k}{3} \tag{5.2}$$

$$Proximity\ threshold = \lceil avg.distance \rceil \tag{5.3}$$

$$Proximity\ threshold\ for\ an\ 8x8\ 2D\ MESH = 2 * \lceil \frac{8}{3} \rceil = 6\ hops \tag{5.4}$$

This close proximity definition can vary according to the size of CMP and various interconnect parameters. However, for simplicity and as a proof of concept, this study considers the average hop latency in 2D Mesh topology as a threshold. As such, Core X is in close proximity of Core Y, if the distance between these two cores are less than the average number of hops needed to go through for a pair of cores inside the corresponding mesh network. For example, close proximity region for core 0 would be seen in Figure 5.6 as colored within the dotted lines in an 8x8 2D MESH interconnect.

5.3 Protocol Implementation Details

In order to sustain correctness and differentiate different scenarios, cache coherence protocols utilize transition states. Each proposed protocol also inherits such transition states from the baseline protocol and uses them to maintain correctness. In essence, for any cache line that does not exhibit producer-consumer sharing pattern, our proposed protocols work like the baseline protocol which is basically enforcing a write-invalidate policy for all cache lines.

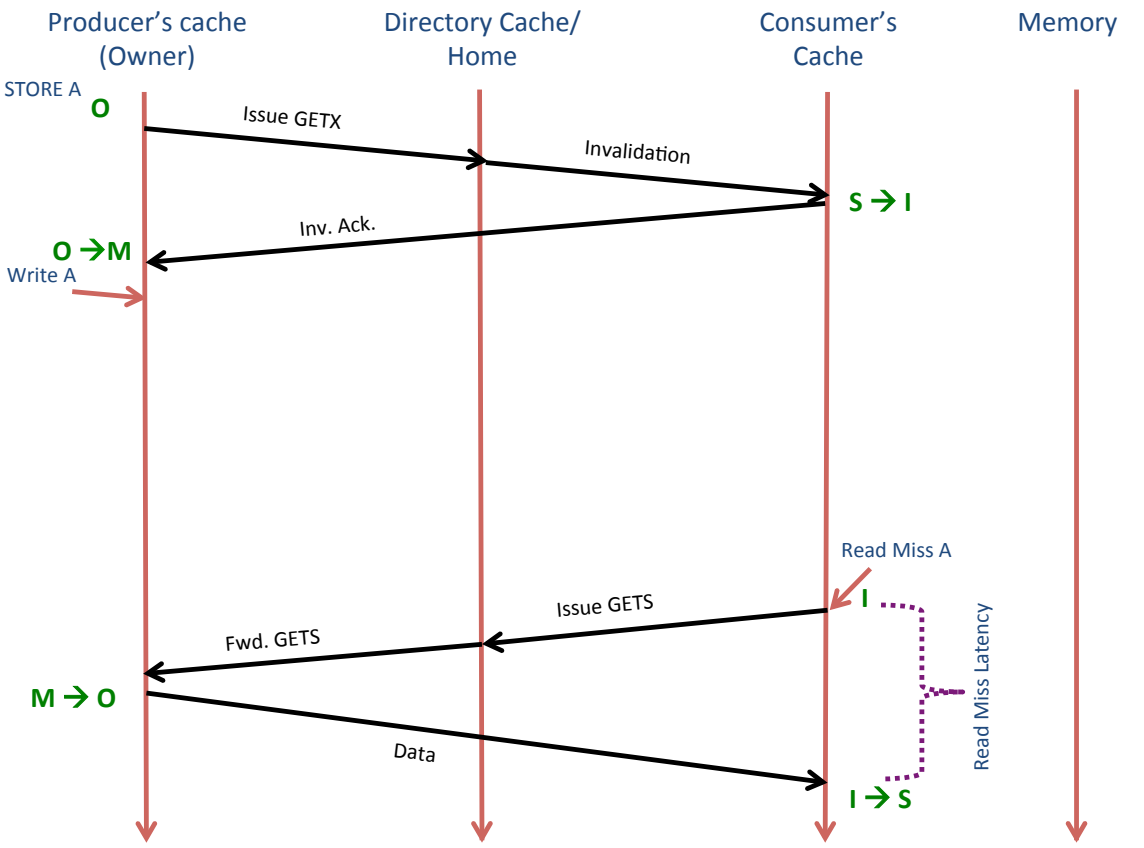


Figure 5.7: Write-Invalidate Directory protocol

All of the proposed protocols can be implemented on top of any write-invalidate directory protocol, but for clarity we gave an overview of our baseline protocol in Section 5.1.2. In addition, Figure 5.7 shows how a typical write-invalidate based protocol works for a sample producer-consumer data sharing example. Each arrow presents a hypothetical time line and designates a specific location in the system. From left to right, these locations are the producer's cache, the home tile which also includes the directory cache, the consumer's cache, and the main memory. The example assumes a single producer and a single consumer for clarity. Also, the cache line is assumed to be loaded and modified earlier so that the cache line actually exists in producer's cache. Similarly, the cache line also exists in the directory cache of the home node/tile. Initial state of the cache line is OWNED in the producer's cache and SHARED in the consumer's cache. The directory cache is inside the home tile and this can be the same tile as the producer. This does not change the order of transactions

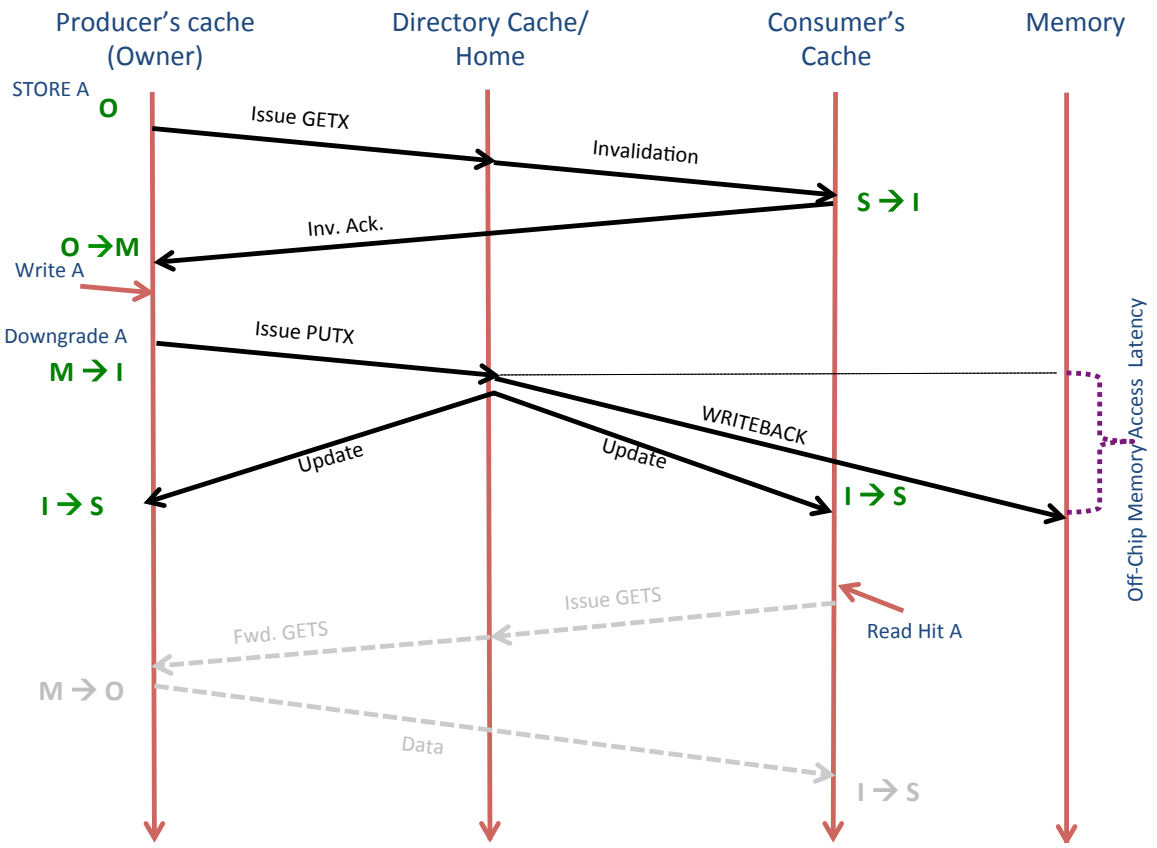


Figure 5.8: Related Work [79]

but the latency of each message, and again for clarity it can be assumed that they represent different cores. Bold lines refer to coherence traffic and dashed gray lines correspond to avoided traffic. Figure 5.7 presents how a write-invalidate protocol would suffer a remote cache miss for a producer-consumer sharing cache line. These remote cache misses can become costly as the number of cores increases. Figures 5.8 and 5.9 present how prediction can help to avoid these coherence misses on the consumers. Figure 5.8 represents an earlier study by Cheng *et al.* [79] and our CMP specific optimizations can be seen by examining Figure 5.9. Our proposed protocols use the same fundamentals to implement a data forwarding mechanism to trigger updates to consumers. However, Figure 5.9 only mentions PCAM for simplicity. In fact, differences between our proposed protocols come from interconnection resource aware mechanisms as explained earlier.

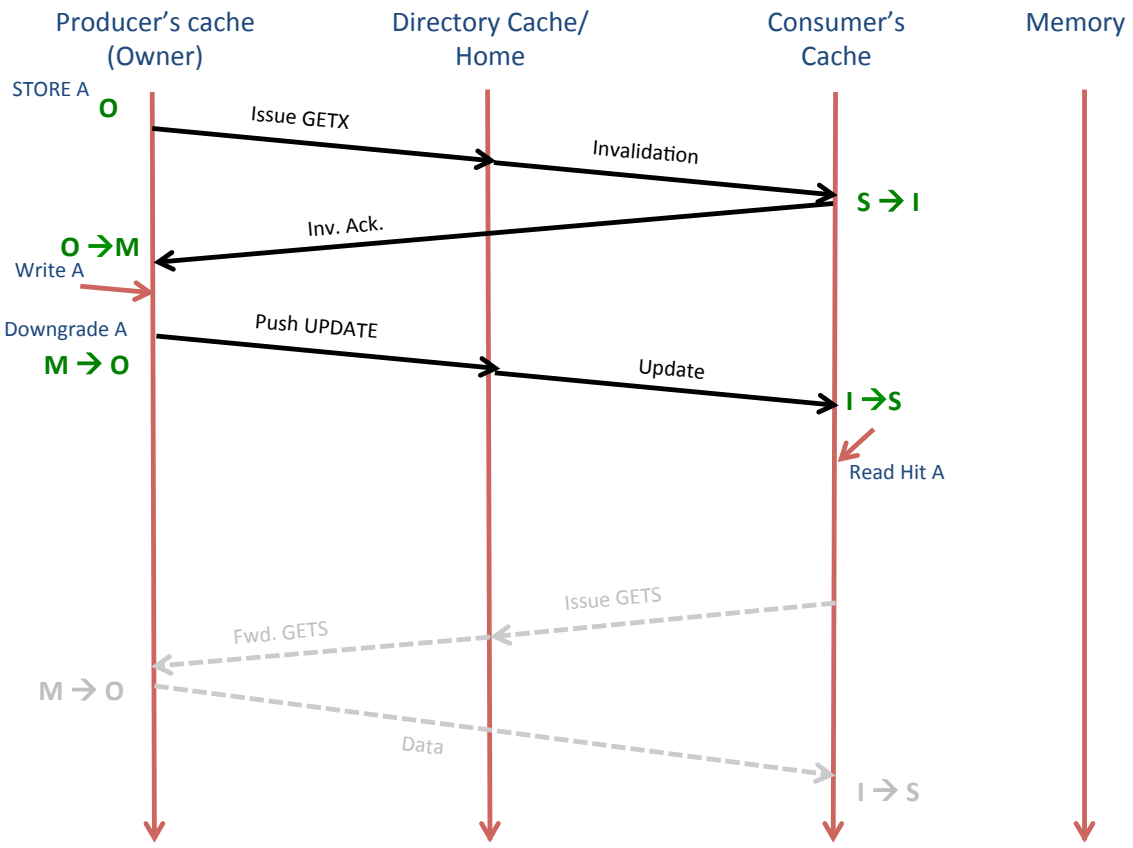


Figure 5.9: Producer-Consumer Adaptive Multicore (PCAM) protocol

Initial transactions in all three examined protocols are same, which presents how the producer gets the exclusive rights on the cache line in order to modify it. With that modification, the cache line becomes MODIFIED in producer's cache, and invalidated in consumer's cache. After this, assuming the prediction mechanism detects that the cache line is actually a producer-consumer sharing line, consumers will be updated in Figures 5.8 and 5.9. However, there is a major difference in how our proposed scheme and the previous study handles the write-update process. Previous work [79] requires a WRITEBACK transaction after the producer modifies the cache line. As such, the cache line is written back to memory and in the meantime the consumer is updated. Even this scheme requires the producer to downgrade its rights on the cache line to be able to do WRITEBACK, and then the home node updates the producer along with the consumer. While this made sense as the processor-to-processor and the processor-to-memory latencies were similar for ear-

lier MPs, this is not the case for multicore chips. Thus, CMPs require further optimizations to avoid off-chip memory accesses. In Figure 5.9, it is shown how PCAM tries to avoid off-chip memory accesses wherever possible to exploit the on-chip resources. In addition, the way the update mechanism is triggered differs between the previous work and PCAM. More details about our trigger update mechanism can be found in the next section.

5.3.1 Trigger Update Mechanism

The *Trigger Update* event is integrated into the baseline cache coherence protocol as a triggering mechanism for the write-updates to consumers. This event becomes active after a certain delay when the producer modifies the producer-consumer cache line. This mechanism is analogous to the delayed intervention mechanism discussed in [79]. However, our trigger update mechanism improves the previous study by adding a dynamic mechanism to trigger updates. In this process, it is important to choose a long enough delay so that producers would finish their write-bursts as producers may write to the cache line more than once after they get the exclusive rights. On the other hand, selecting a long delay might be too late for consumers and would not be able to prevent a cache miss in consumer's cache. Thereby, we did a sensitivity analysis which can be seen in Figure 6.6 in order to select the best value. Accordingly, a static 1000 cycle latency is selected to trigger the update mechanism after the producer modifies a cache line with producer-consumer sharing pattern. However, this value should be configured based on the specific hardware configuration and system settings during the design stage as the optimal value may differ from the one chosen for our specific system.

As an optimization to this static trigger update mechanism, we devised a novel dynamic scheme that can adapt to the consumer's behavior in such update scenarios. This dynamic trigger mechanism still uses the static trigger delay, however, if there is a GETS (Get

Shared) request from one of the consumer's before the delay is ended, not only the GETS request is answered with the corresponding data block but also *Trigger Update* event is called to push updates to other consumer(s). In the meantime, this cancels the trigger delay to prevent unnecessary future updates. If the request comes from a core that is at the point not a member of the consumer's list, the requesting core is added the consumer list and updates are still pushed to the consumer(s). As such, this dynamic scheme can improve the performance of the update mechanism by preventing late triggers that fail to prevent cache misses in consumers. The trigger update mechanism goes back to the static case, where it waits for 1000 cycles, after each update. There is no difference between the static and dynamic schemes if GETS request comes after the trigger delay period. In fact, that scenario is our goal which results in a cache hit in consumer's cache.

5.4 Experimental Verification of the Protocols

Our proposed cache coherence protocols rely on the baseline coherence protocol for correctness. However, extensions made to the base protocol can easily create race conditions which need to be verified and tested. Although formal verification of our proposed protocols is a ongoing work, we utilized stress tests provided by GEMS. This synthetic testing mechanism which can generate excessive race conditions on the cache coherence protocol helped us to identify the potential coherence bugs. Furthermore, we managed to run all benchmark suites to completion with successful data verification at the end.

5.5 Analytical Model

In this section, we introduce an analytical model to assess the performance of parallel applications under our proposed adaptive cache coherence protocols. In general, this model is intended to provide more insights in evaluating the performance of PCAM versus the baseline directory-based write-invalidate protocol. The model is adapted from an earlier study by Srbljic *et al.* [134]. As such, the model uses similar access patterns and parameters. However, we extended their core model to analyze our proposed protocols. Since, our interconnection-resource-aware protocols, PCBAM and PCPAM, are actually extensions of PCAM. We only use PCAM in the analytical model study to compare against the base write-invalidate directory protocol. Our goal is to have a simple intuitive model to be able to assess potential performance gains from a dynamic protocol such as PCAM and eventually better interpret the results obtained from our experimental studies. As such, the parameters required in our model can easily be obtained from a memory trace of the application via a simulator, hardware counters or even at compile time with the help of an advanced compiler. In addition, this analytical model can be used to investigate scalability studies for larger CMPs as simulations are not feasible after some point due to timing and infrastructure restrictions.

Table 5.2: Data Access Patterns and Parameters

Patterns		Parameters	
MR	Multiple Reader	β	number of unique processors that perform reads
MW	Multiple Writer	β	number of unique processors that perform writes
SRSW[a]	Single Reader Single Writer	α	probability that the access is a write
		$1-\alpha$	probability that the access is a read
MRMW[b]	Multiple Reader Multiple Writer	β	number of unique processors that access the block
		$\frac{\alpha}{\beta}$	probability that the access is a write from a specific one of the β processors[c]
		$\frac{1-\alpha}{\beta}$	probability that the access is a read from a specific one of the β processors[d]

[a]Single Reader (SR) and Single Writer (SW) patterns are also part of SRSW with α set to 0 or 1, respectively.

[b]Multiple Reader Single Writer (MRSW) and Single Reader Multiple Writer (SRMW) patterns are assumed to be part of the MRMW pattern

[c]Probability is calculated assuming each one of the β processors perform each access with equal probability.

[d]Probability is calculated assuming each one of the β processors perform each access with equal probability.

Table 5.3: System Events

Event	Description
E_1	Read a data block from memory
E_2	Read a data block (Dirty copy) from remote cache
E_3	Read one word from local cache (Cache Read HIT)
E_4	Obtain ownership and invalidate other copies
E_5	Read a data block from memory and modify it
E_6	Read a data block (Dirty copy) from remote cache and modify it
E_7	Write one word to local cache (Cache Write HIT)
E_8	Write back a data block to memory

Table 5.4: Probabilities and Corresponding Costs for System Events for the MRMW Pattern

Load Events	
Probabilities[ab]	Cost
$\phi_1 = \frac{\alpha(\beta-1)(1-\alpha)}{1+(\beta-1)\alpha} - \frac{\alpha(\beta-1)(1-\alpha)}{\alpha+\beta-1}$ $\phi_2 = \lambda(\frac{\alpha(\beta-1)(1-\alpha)}{\alpha+\beta-1})$ $\phi_3 = \eta(\frac{(1-\alpha)}{1+(\beta-1)\alpha})$ $\sum_{i=1}^{3} \phi_i = 1 - \alpha = p(read\ access)$	$c_1 = 100$ $c_2 = 24 + ((t_s + t_h) * (avg.\ \#\ of\ hops))$ $c_3 = 0$[c]
Store Events	
$\phi_4 = \alpha - \frac{\alpha^2(\beta-1)}{1+(\beta-1)\alpha} - \frac{\alpha^2}{\alpha+\beta-1}$ $\phi_5 = \lambda(\frac{\alpha^2(\beta-1)}{1+(\beta-1)\alpha} - \frac{\alpha^2(\beta-1)}{\alpha+\beta-1})$ $\phi_6 = \lambda(\frac{\alpha^2(\beta-1)}{\alpha+\beta-1})$ $\phi_7 = \sigma(\frac{\alpha^2}{\alpha+\beta-1})$ $\sum_{i=4}^{7} \phi_i = \alpha = p(write\ access)$	$c_4 = 24 + ((t_s + t_h) * (avg.\ \#\ of\ hops))$ $c_5 = 100 + ((t_s + t_h) * (avg.\ \#\ of\ hops))$ $c_6 = 24 + ((t_s + t_h) * (avg.\ \#\ of\ hops))$ $c_7 = 0$[d]

[a] λ, η, and σ are considered to be 1 for the baseline directory protocol.

[b] Probability ϕ_8 is zero since we assume infinite caches and all the formulas are based on a steady-state analysis

[c] Cache read hit is assumed to have zero cost.

[d] Cache write hit is assumed to have zero cost.

5.5.1 Assumptions

The analytical model is observing the fact that the memory space is partitioned into data blocks of equal size, such as cache lines. The model is then built by associating each data block with an access pattern that will be further explained in the next subsection. A list of the data access patterns and the parameters used with these patterns are demonstrated in Table 5.2. To obtain the initial parameters, we use the generated memory traces of our workloads. The cost related parameters are representing the values for our target tiled-CMP and can easily be modified to reflect technological changes. On the other hand, application dependent parameters can be extrapolated from these initial calculated values for larger CMPs. Data access patterns are obtained by tracking the number of cores that accesses the block and the type of accesses (read or write). The parameters associated with each access pattern include the number of cores that accesses the block, and the probabilities

of the type of accesses. Based on these information, analytical formulas are derived to predict the steady-state cost of running a parallel application with a given cache coherence protocol. While deriving the formulas and calculating the costs, accordingly we make some assumptions that are summarized as follows:

- The memory space is partitioned into data blocks of equal size as in a typical memory trace obtained by simulation.
- Events are assumed to be independent, and each access type has equal probability of being executed on any of the processors that access the given data block.
- No capacity misses at the caches as they are assumed to be infinite size similar to previous models [88, 134, 135] to simplify derivations and focus on cache coherence traffic.
- Cache read/write hits have zero cost.

5.5.2 The CMP system model

As we described earlier, our proposed protocols are based on a write-invalidate protocol. As such, the performance results compare our proposed protocols against a baseline protocol, which is a write-invalidate based directory protocol. Also, we use write-back private caches in our modeled tiled-CMPs. Therefore, we first identify possible system events that can happen in a CMP system with write-invalidate, write-back caches. These events are shown in Table 5.3. Our formulas associate each event with a cost and probability to predict the performance of the protocol for a given parallel application.

5.5.3 Data Access Patterns and Parameters

Our formal model uses same access patterns and parameters as suggested earlier by [134]. However, we combine SRMW and MRSW patterns under MRMW as they can all be considered as producer-consumer sharing patterns and as mentioned earlier our goal is to come up with a simple, easy to obtain model to assess potential performance gains from PCAM rather than a more accurate performance prediction model. Table 5.2 illustrates the considered data access patterns along with the corresponding parameter(s) utilized to calculate the overall cost of pattern and eventually the whole application. Number of unique processors accessing the data block is denoted by the parameter β. Single Reader (SR) and Single Writer (SW) patterns are also part of SRSW with α set to 0 or 1, respectively.

5.5.4 Cost Calculation of Data Access Patterns

Probability of a write access is denoted by the parameter α, and the read probability $(1-\alpha)$ is derived accordingly as an access can be either a read(load) or a write(store) operation. Also, each processor is assumed to have equal chance to access the data block. Accordingly, the probability of W_i (processor i writing to the block) would be $\frac{\alpha}{\beta}$ when β processors access the data block.

Accordingly, the average cost in terms of cycles of accessing a cache line with a specific pattern for a given protocol can be calculated using the formula seen in Equation 5.5. Protocol refers to either the baseline directory protocol or PCAM. Patterns are given in Table 5.2. Parameter c_k refers to the cost of event E_k and ϕ_k refers to the probability of event E_k. Finally, the average cost per shared access for the entire application becomes as it is shown in Equation 5.6. Here the c_k parameters are selected to capture hardware properties, while the ϕ_k parameters are selected to represent application behavior.

$$C_{pattern_x,protocol} = \sum_{E_k} c_k \phi_k \tag{5.5}$$

$$C^{application}_{protocol} = \sum_{line\ x} \frac{\#\ Accesses_x}{\#Accesses} C_{pattern_x,protocol} \tag{5.6}$$

The probabilities ϕ_n for a given access pattern is derived similar to the study [134]. To summarize, events are assumed to be independent so a probability of a sequence is the product of all the probabilities of the accesses involved in that sequence. To obtain the probability of a specific system event, the probabilities of all sequences which result in this event have to be summed. We introduce λ and η as new parameters to analyze PCAM and differ from the baseline cache coherence protocol. We provide some of the derivations of formulas for probabilities that present how we calculated the system event probabilities given in Table 5.4. For example, system event E_2 refers to a read operation of a modified data block from a remote cache. First we identify, the sequence of operations that represent such a system event, such as $W_i(R_i, \cdots, R_i)R_j$ for E_2. R_i and R_j denote read operations from cores i and j, and W_i refer to a write operation from core i. To obtain the summation of the probabilities for all possible pairs, we need to multiply the probability of this sequence with $\beta(\beta - 1)$. Accordingly, the derivation of ϕ_2 can be done as follows:

$$\begin{aligned}
\phi_2 &= (\beta)(\beta-1)p(\sum_{m=0}^{\infty} W_i \underbrace{(R_i, \cdots, R_i)}_{m\ times} R_j) \\
&= (\beta)(\beta-1)\sum_{m=0}^{\infty} p(W_i)(p(R_i))^m p(R_j) \\
&= (\beta)(\beta-1)\sum_{m=0}^{\infty} (\frac{\alpha}{\beta})(\frac{1-\alpha}{\beta})^{m+1} \\
&= \frac{\alpha(\beta-1)}{\alpha+\beta-1}
\end{aligned} \tag{5.7}$$

System event E_6, is an exclusive read from a remote cache which can be represented with a sequence of the form, $W_i(R_i, \cdots, R_i)W_j$. It presents the scenario where write from core i is followed by a write from a different core j, this includes scenarios where core i reads the block any number of times before core j actually writes to it. This probability can be calculated as shown in Expression 5.8.

$$\begin{aligned}
\phi_6 &= (\beta)(\beta-1)p(\sum_{m=0}^{\infty} W_i \underbrace{(R_k, \cdots, R_k)}_{m\ times} W_j) \\
&= (\beta)(\beta-1)\sum_{m=0}^{\infty} p(W_i)(p(R_k))^m p(W_j) \\
&= (\beta)(\beta-1)\sum_{m=0}^{\infty} (\frac{\alpha}{\beta})^2(\frac{1-\alpha}{\beta})^m \\
&= \frac{(\beta-1)\alpha^2}{\alpha+\beta-1}
\end{aligned} \tag{5.8}$$

Chapter 6

Performance Results for the Producer-Consumer Adaptive Cache Coherence Protocols

This chapter presents the performance evaluation for our proposed cache coherence protocols. We first describe the experimental plan including the simulation environment and the workloads used. Then, we present our experimental and analytical performance investigation research. We compare the experimental results with the analytical model for verification.

6.1 Experimental Plan

This section presents our experimental study; simulation environment, workloads and performance evaluation of our proposed protocols.

6.1.1 Simulation Environment

This research is conducted with full-system simulation utilizing Virtutech Simics [27] extended with a modified version of the GEMS toolset [125]. A full-system simulator was chosen to include operating system effects to establish a more practical platform for our experiments. Simics provides the functional full-system simulation and GEMS provides a detailed memory system timing infrastructure which observes all cache coherence protocol messages and the corresponding state transitions. GEMS includes a module, Ruby, to model memory hierarchies. Ruby uses the SLICC (Specification Language for Implementing Cache Coherence) domain-specific language to specify different cache coherence protocols. As such, we used SLICC to implement our proposed cache coherence protocols. The simulated architecture is a many core SPARC tiled-CMP system, running unmodified Solaris 10. Table 5.1 provides the details of the simulation environment along with the target architecture details.

As full-system simulation suffers slowdowns of several orders of magnitude, which gets even worse with the increasing number of cores, we provide results mainly from a 16-core tiled-CMP. We also show results from a 64-core tiled-CMP with limited number of workloads. In addition, trace-based simulation and analytical model are used to provide results from even a larger CMP up to 256 cores. Caches are warmed up to avoid cold misses by utilizing the checkpointing functionality provided by Simics. Accordingly, the results are reported from the parallel computational phase of each workload, but the workloads are run to completion.

6.1.2 Simulation Workloads

The benchmarks for the full-system simulation study were selected from the SPLASH-2 [117] benchmark suite and Omni's OpenMP version of the NAS Parallel benchmark (NPB) suite [136, 137]. The input data sets for the simulation workloads are provided in Table 6.1.

Table 6.1: Simulation Workloads

Benchmark	**Input Data Set**
NPB BT	12*12*12 nodes, # of Iterations=60
NPB CG	Size=1400, # of Iterations=15
NPB LU	16*16*16 nodes, # of Iterations=50
Barnes	1024 nodes, 123 seed
Cholesky	input tk29.O
FFT	64K complex data points
FMM	16K particles
Ocean_cont	258*258 grid, 1e-7 error tolerance
Ocean_noncont	258*258 grid, 1e-7 error tolerance
Raytrace	car.env

Barnes is an implementation of the Barnes-Hut method to simulate the interaction of a system of bodies over time in a gravitational system (N-body problem). BT solves a partial differential equation using a block tridiagonal algorithm. CG implements unstructured matrix vector multiplication via conjugate gradient method to approximate the smallest eigenvalue of a large, sparse, symmetric positive definite matrix. Cholesky kernel performs a blocked cholesky factorization on a sparse matrix. FFT benchmark solves a complex, one-dimensional version of the $radix - \sqrt{n}$ six-step FFT. FMM application implements a parallel adaptive Fast Multi-pole method to simulate the N-body problem like Barnes. LU benchmark refers to the LU decomposition. Ocean application is a simulation of large-scale ocean movements based on eddy and boundary currents. We used both of the contiguous and non-contiguous versions of the Ocean code. The non-contiguous version implements the grids in two-dimensional arrays which results in non-contiguous memory allocation.

On the other hand, the contiguous version implements the grids with three-dimensional arrays. First dimension is used to specify the local processor for a given partition and accordingly allows contiguous allocation of particles inside the local memory of processors that own them. The Raytrace application implements an algorithm to render a three-dimensional scene onto a two-dimensional image plane using optimized ray tracing.

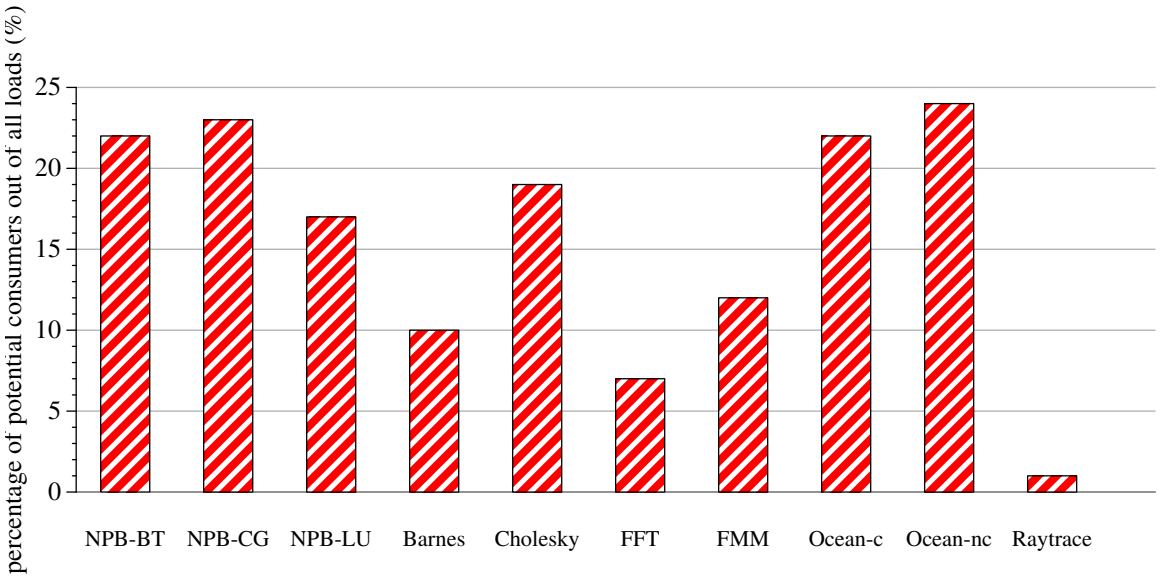


Figure 6.1: Ratio of potential consumers out of all LOADs

6.2 Cache Coherence Protocol Performance Results

In this section, we present the performance results from the workloads presented in Table 6.1. For all the figures; PCAM refers to our Producer-Consumer Adaptive Multicore protocol, PCBAM refers to our Producer-Consumer Bandwidth Adaptive Multicore protocol, PCPAM refers to our Producer-Consumer Proximity Adaptive Multicore protocol, and base refers to the baseline write-invalidate directory-based cache coherence protocol, as described in Section 5.1.2. As mentioned earlier in Section 5.3.1, two different trigger

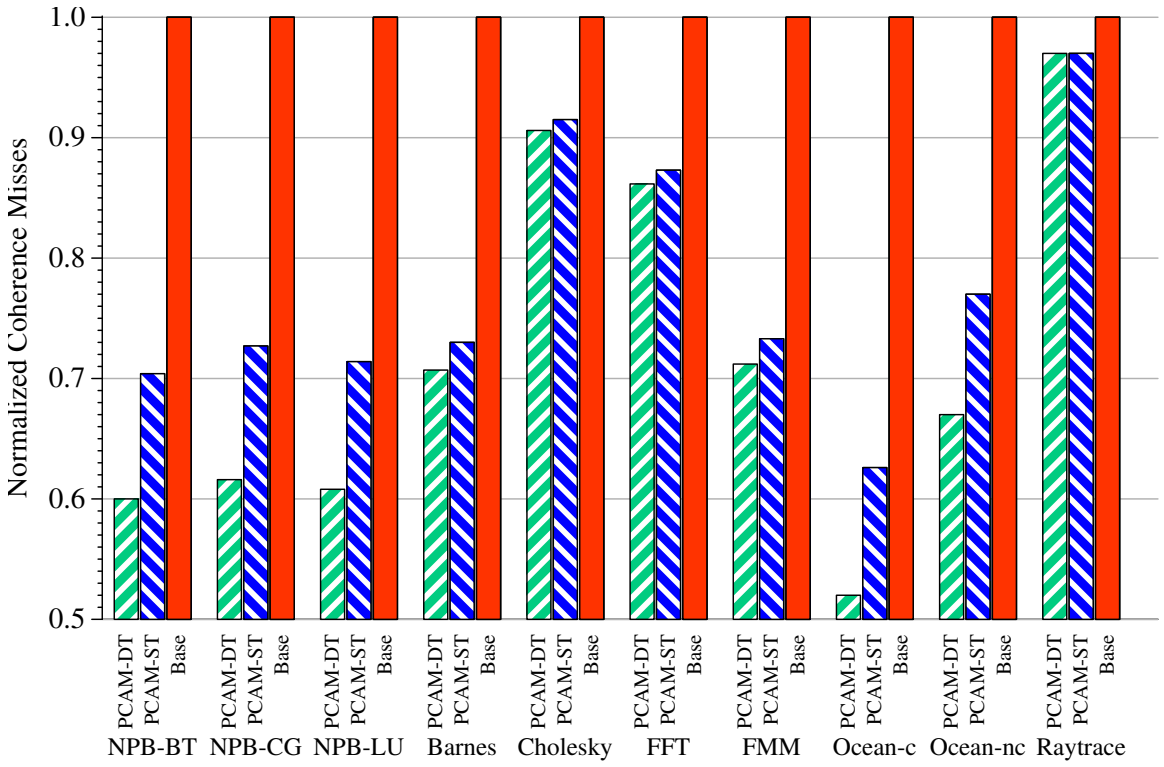


Figure 6.2: Cache coherence misses improvement

update mechanisms (static vs. dynamic) are utilized in our experiments. To differentiate them, for example, they are represented as PCAM-ST (static) and PCAM-DT (dynamic) for the PCAM protocol results.

Accordingly, Figures 6.2 and 6.3 show the performance comparison between our proposed adaptive, hybrid protocol (PCAM) and the baseline protocol. PCAM-ST is experimented with 1000-cycle Trigger Update latency. PCAM-DT also uses the same Trigger Update latency but dynamically triggers updates to consumers if the producer receives a GETS request earlier than this 1000 cycle static threshold. Results show improvements up to 48% with an average of 31% for coherence misses as illustrated in Figure 6.2. In general, our proposed optimizations provide performance gains proportional to the available producer-consumer sharing pattern inside the given workload. As such, Figure 6.1 presents the percentage of LOADs that are part of a producer-consumer sharing pattern to better interpret

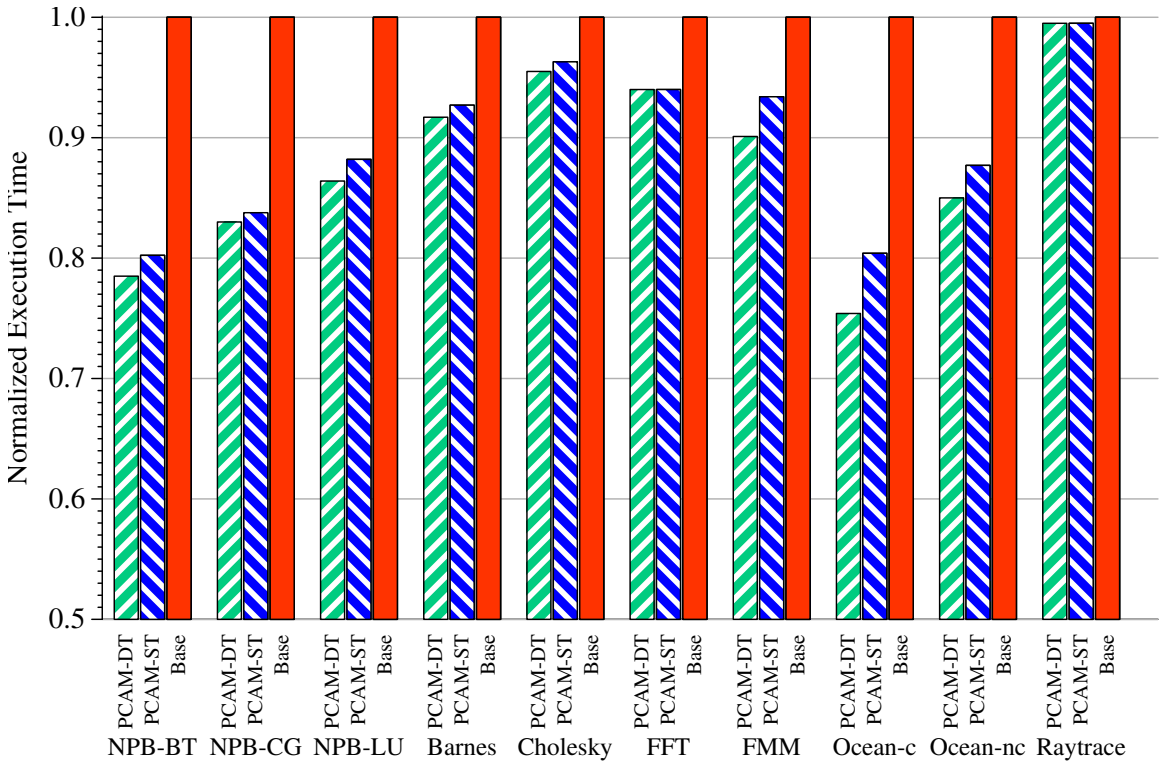
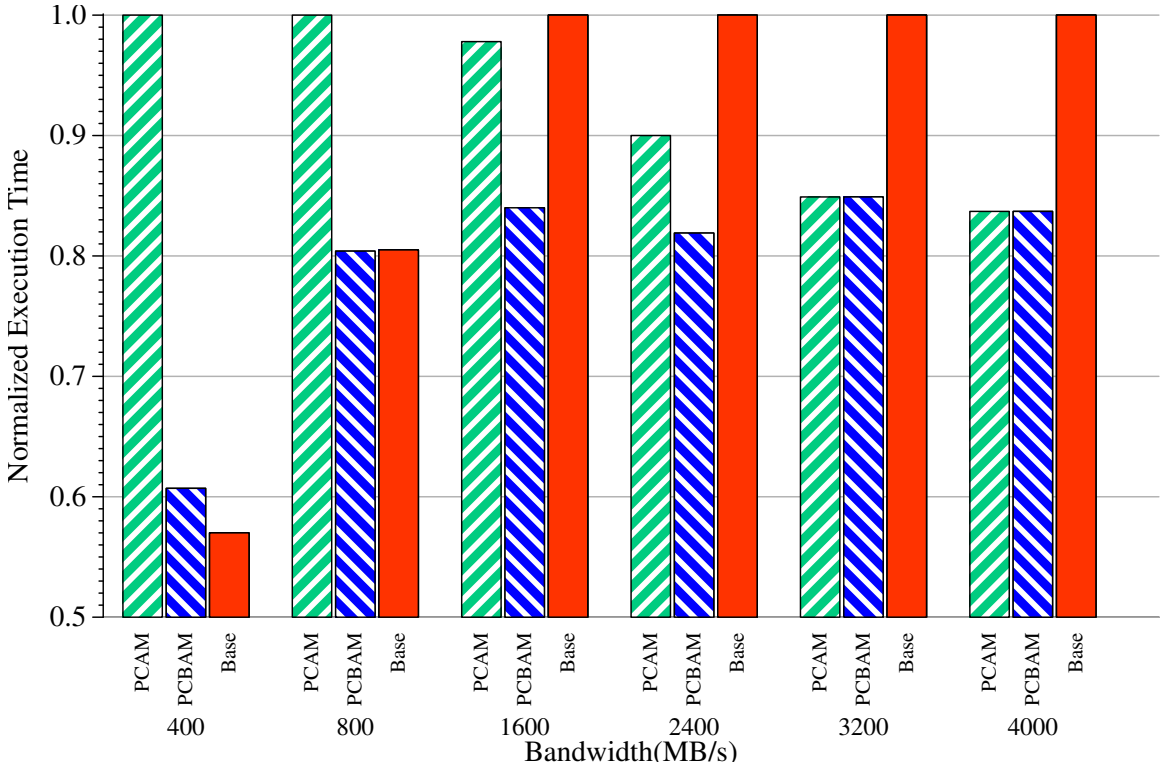


Figure 6.3: Execution time improvement

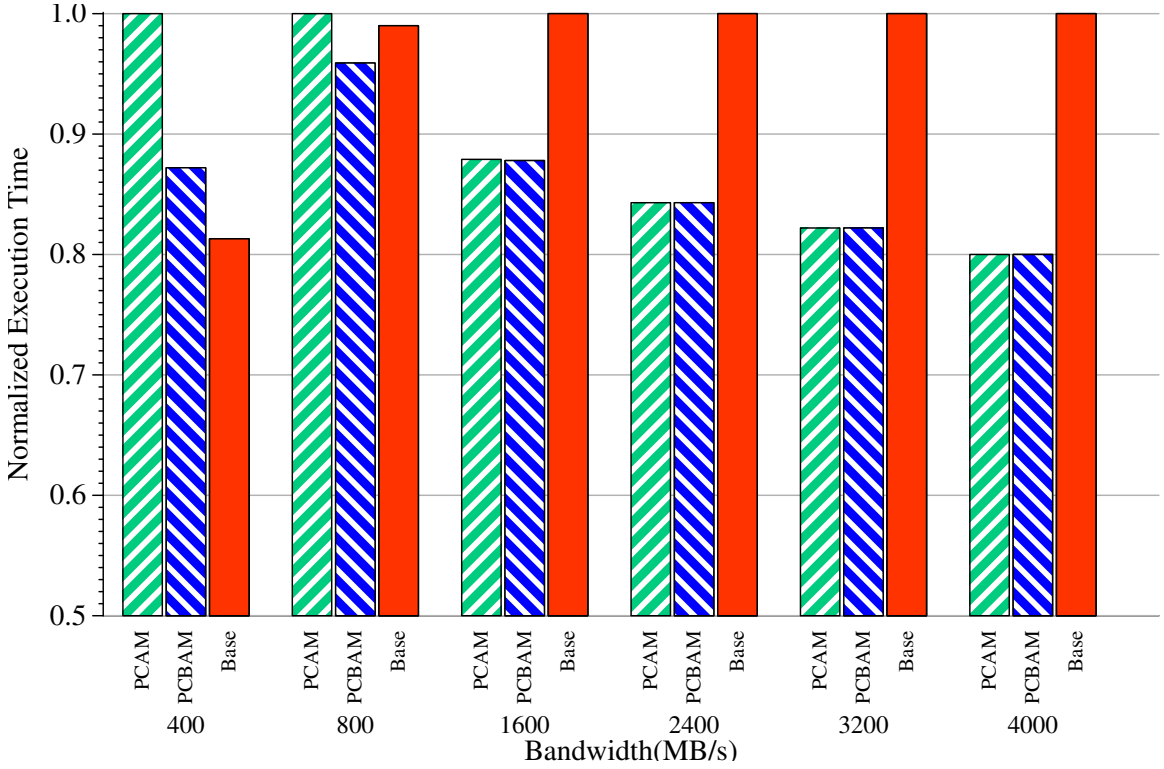
the performance results. This profiling analysis is done by post-processing the memory traces of the applications obtained from our simulation framework. Total execution time improvements may vary for different architectural settings, but results demonstrate performance gains in all the applications experimented. Since we manage to reduce the coherence misses, these performance gains should even be higher for larger number of cores inside the chip where remote miss latencies will have more severe affects.

6.2.1 Bandwidth Adaptivity Analysis

Figure 6.4 shows results from our performance evaluation of the bandwidth adaptive mechanism. Results from four different workloads; NPB CG, NPB LU, Ocean, and Cholesky are presented with varying link bandwidth characteristics. Performance results for each

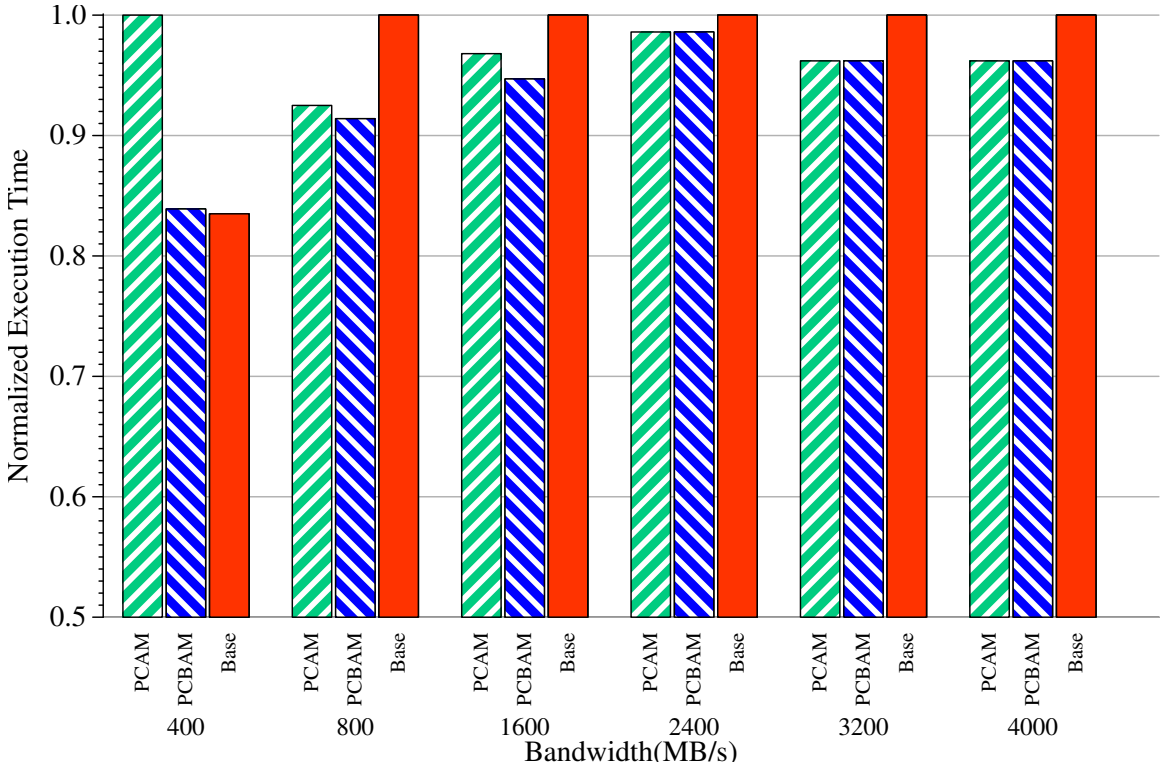


(a): NPB CG

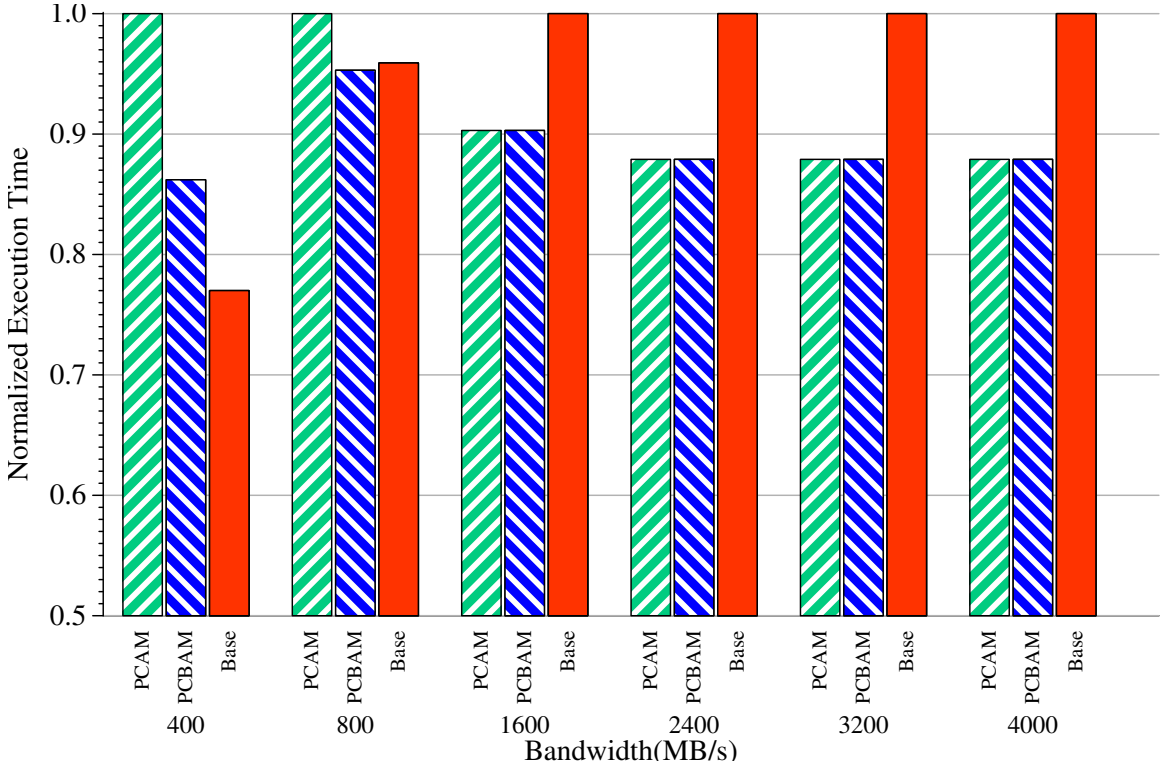


(b): Ocean cont.

Figure 6.4: Bandwith adaptivity results



(c): Cholesky



(d): NPB LU

Figure 6.4: Bandwith adaptivity results

link bandwidth value is normalized to the worse result for that specific link bandwidth. Experiments vary the static link bandwidth between 400 MB/s to 4000 MB/s. One thing to note here is the fact that our simulator is based on an in-order SPARC core. Accordingly, bandwidth utilization in such a system is lower than the many contemporary systems. However, CMPs with larger number of cores will have relatively lower bandwidths compared to the current industrial trends. To compensate this, we vary our tests in a way to throttle the links in terms of bandwidth to create a scenario that can happen with a more aggressive out-of-order processor under larger link bandwidths. The goal is to check whether PCBAM can help when there is a network congestion in the system by dynamically adapting and cutting off any update optimizations which tend to generate more traffic than simple invalidates. Comparison of PCAM, PCBAM and base protocols clearly indicate how PCBAM can adapt to low bandwidth situations and mitigate any potential performance loss. Results show that a hybrid/adaptive protocol such as PCAM without a bandwidth-aware mechanism can suffer a performance degradation with high interconnect utilization. PCBAM performed up to 40% better in low-bandwidth scenarios as seen for the NBP CG case. Furthermore, when bandwidth is plentiful PCBAM performs similar to PCAM as the bandwidth-adaptive mechanism mostly lets update optimizations to be triggered in such cases.

6.2.2 Proximity Aware Protocol Analysis

We also extended our producer-consumer adaptive protocol with a proximity-aware mechanism. The proximity-aware mechanism observes the importance of spatial locality in future CMPs. As such, with this mechanism we establish latency based updates to consumers. It is based on the assumption that updates to far remote caches can be costly in terms of network traffic and accordingly may degrade performance. With this in mind, the proximity-aware mechanism is implemented on top of the PCAM protocol, and it is

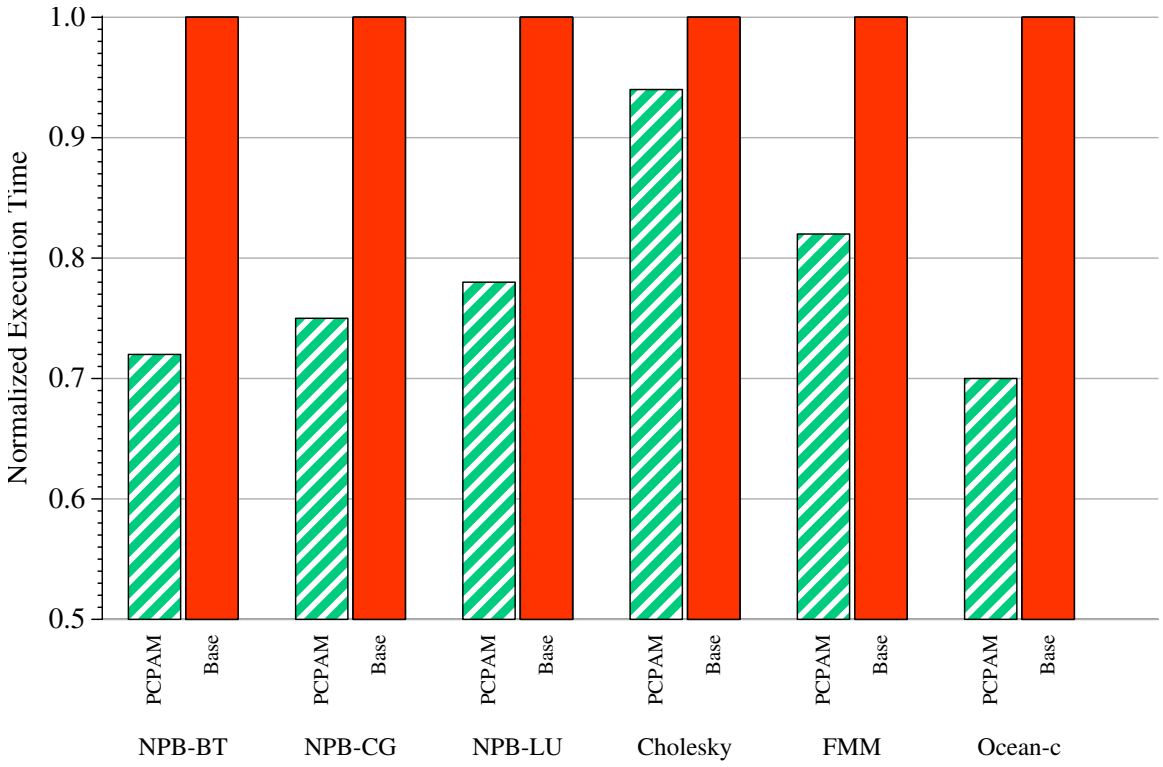
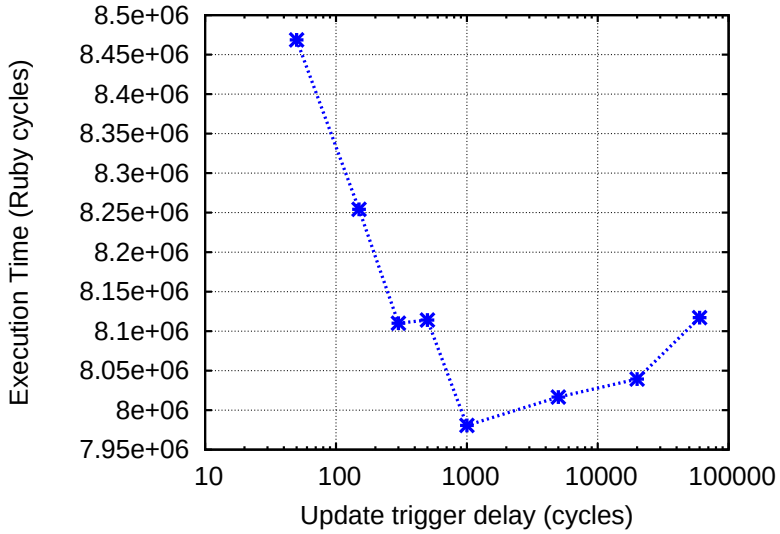
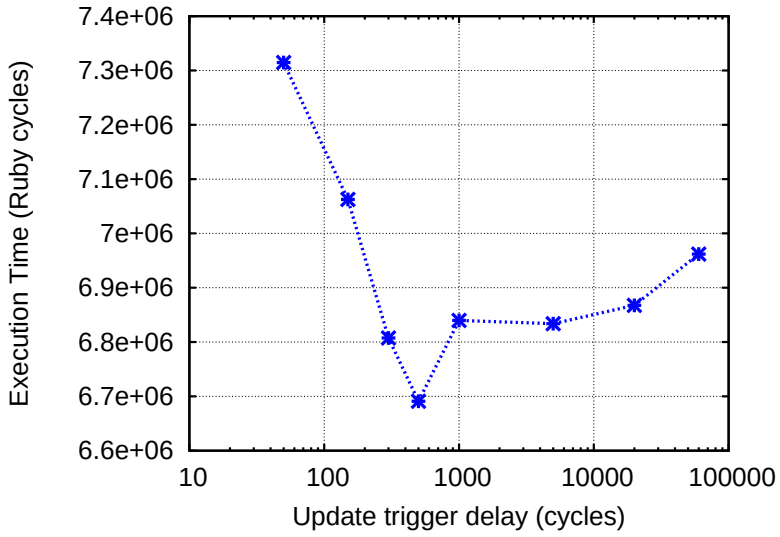


Figure 6.5: Performance Results from a 64-core tiled-CMP

referred to as "*Producer-Consumer Proximity Adaptive Multicore (PCPAM)*" protocol in this work. PCPAM pushes updates to consumers only within a certain proximity of the producer. This proximity selection was discussed earlier in Section 5.2.4, which equals to the average distance in the network. Furthermore, PCPAM uses the dynamic trigger update mechanism. As such, in normal cases PCPAM pushes updates to consumers only with in the proximity of the producer. However, when a GETS arrives from outside this proximity, PCPAM switches to PCAM like updates and the protocol pushes the updates to all consumers in the system. Figure 6.5 show the corresponding results that are provided from a 64-core tiled-CMP platform. Results present a smaller workload set due to the excessive simulation times incurring from 64-core full-system simulations. It is noted that the performance improvements increase as the core counts increase.

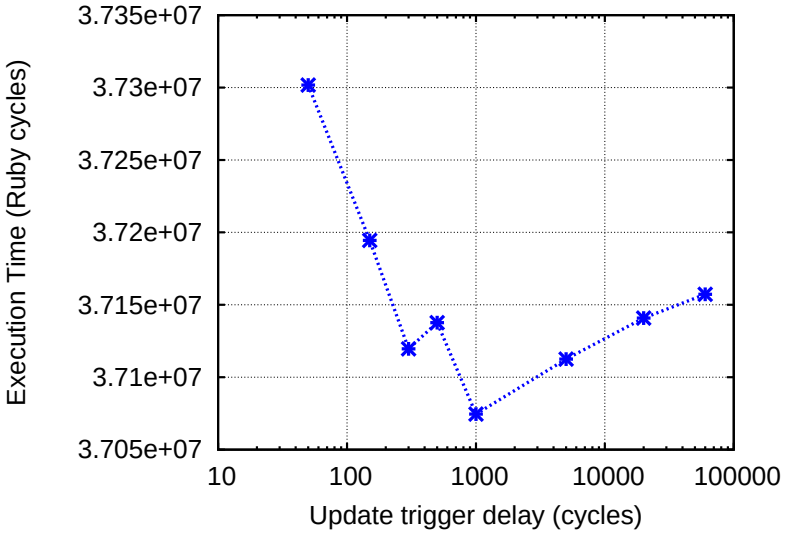


(a): Barnes

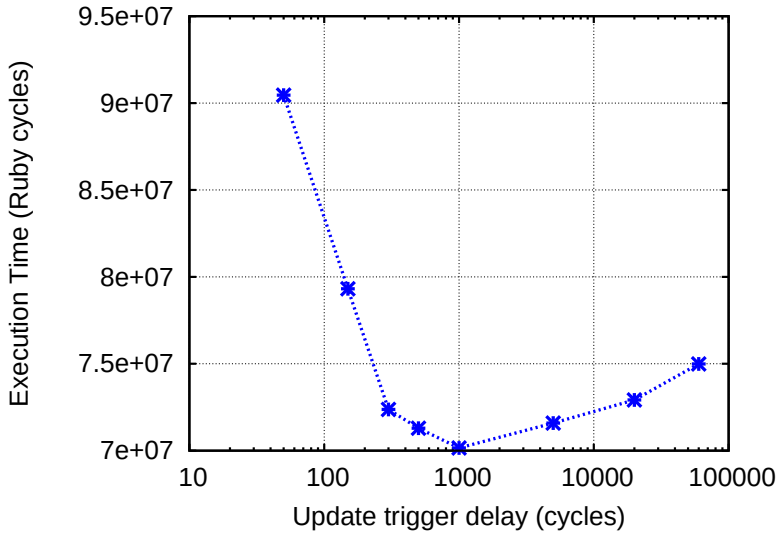


(b): FMM

Figure 6.6: Update Trigger delay sensitivity analysis



(c): LU



(d): Ocean cont.

Figure 6.6: Update Trigger delay sensitivity analysis

6.2.3 Sensitivity Analysis

6.2.3.1 Trigger Update Delay Latency Sensitivity Analysis

In order to select the Trigger Update delay latency, we conducted a sensitivity analysis for a wide range of delay cycles while running our workloads. Corresponding results can be seen in Figure 6.6. The y axes in the figures refer to the total execution time which is presented as Ruby cycles. Ruby is the detailed memory timing simulator inside GEMS. On the x axes, different values, in cycles, for the update trigger delay are shown. Although there are slight variations, 1000 cycles is proven to be an optimal value for most of the cases. However, some workloads tend to be insensitive for a wide range of delay cycles. Hence, they are not included in the results. In addition, our *dynamic trigger update* mechanism further optimizes this process, and makes the choice of 1000 cycles much safer.

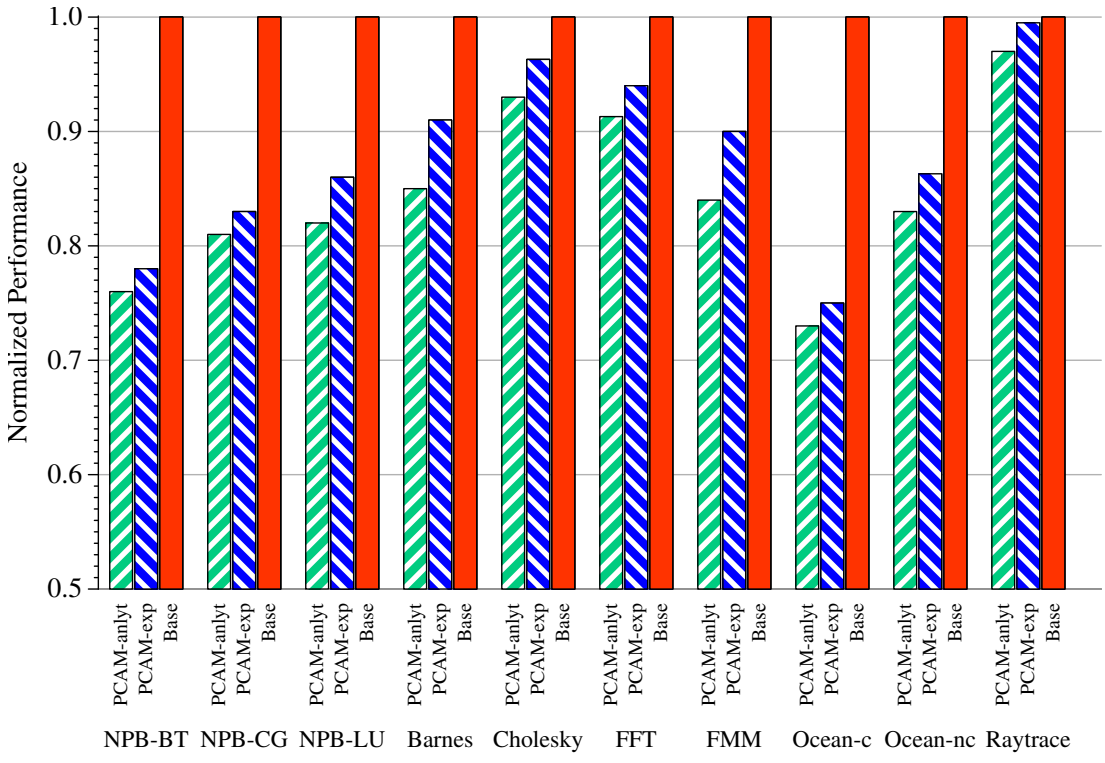


Figure 6.7: Performance Results with Analytical Model

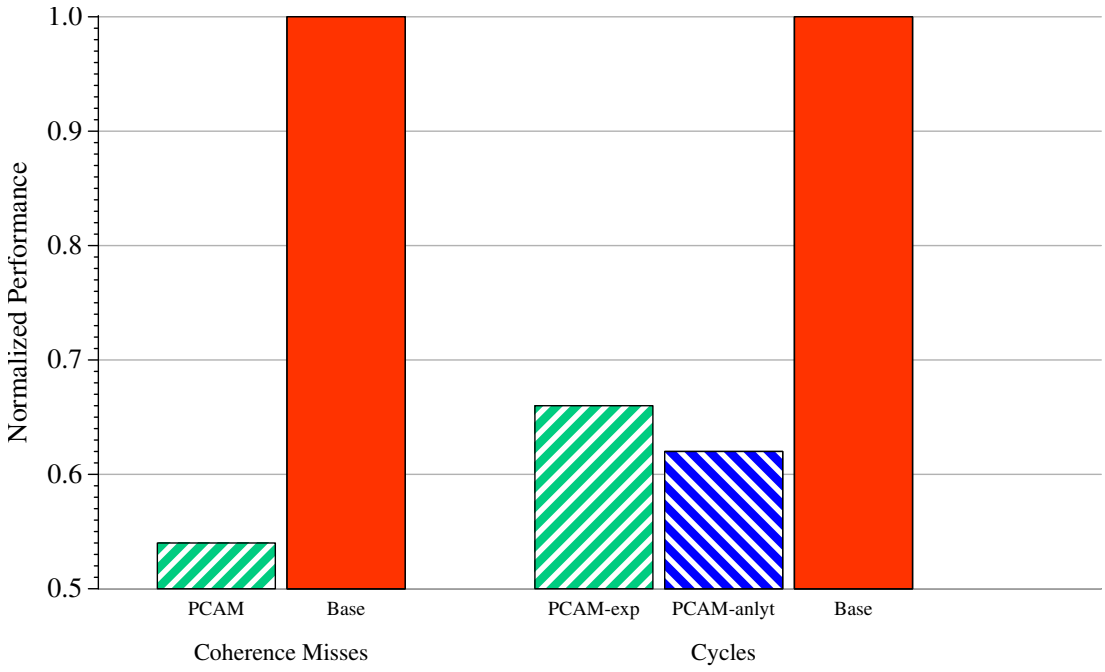


Figure 6.8: Performance Results from 256-core tiled-CMP

6.2.3.2 Analytical Model Results

Here we present the results from the aforementioned analytical model. The parameters needed for the analytical model are obtained through analyzing the memory traces of each workload. During our simulation experiments, we also collected the memory traces of the running workloads to be used in this study. Figure 6.7 compares the execution time results from the experimental measurements using the simulators (base and PCAM-DT) with the values obtained by using our analytical model. *PCAM-exp* and *Base* represent simulation results, whereas *PCAM-anlyt* presents the results from the analytical model. It is shown that the analytical model can estimate the performance gains from our proposed protocol, PCAM, with an error margin of 2.2% up to 6.5%, which is an acceptable accuracy given the value for the model especially for large core counts. In addition, we generated a synthetic micro-benchmark trace, which exhibits producer-consumer sharing pattern. Utilizing only the GEMS toolset, we created a sample 256-core tiled-CMP with again a 2D-MESH in-

terconnect. Figure 6.8 presents the results from this study by showing both the coherence misses and execution time analyses. It can be seen that 46% improvement in coherence misses turn into a 34% improvement in the total execution time. In addition, we utilized the analytical model to estimate the performance of this synthetic benchmark, and the result shows a 38% performance improvement as estimated by the model.

Chapter 7

Summary and Conclusions

7.1 Summary

The transition to multicore processors provides new ways of achieving performance gains mainly through Thread-Level Parallelism (TLP). TLP requires parallel programming, and accordingly most Chip Multiprocessors (CMPs) have adapted the shared-memory model for ease of use. The shared memory model entails communication primitives such as cache coherence and synchronization mechanisms, in order to maintain correctness while concurrently running multiple threads. Such mechanisms in CMPs can benefit from the earlier research studies for conventional MPs. However, treating CMP based systems as conventional MPs would miss significant new opportunities as CMPs offer particular characteristics that are fundamentally different from traditional MPs. Cache coherence protocols have become a first-order design issue at the chip level. The cache coherence mechanisms are a key component in achieving performance growth in CMP based systems through TLP and therefore, this calls for cache coherence protocol designs considering CMP specifics. Thereby, the main goal of this research is to analyze and optimize CMPs from a cache

coherence perspective.

In accordance with our observations from the preliminary experimental work on performance analysis, we have formally modeled the architectural components for major performance bottlenecks using a well-known simulator infrastructure focusing mainly on cache coherence. This research is conducted with full-system simulation utilizing Virtutech Simics extended with the GEMS toolset. A full-system simulator was chosen to include operating system effects to establish a more practical platform for our measurements. Simics provides the functional full-system simulation and GEMS provides a detailed memory system timing infrastructure which observes all cache coherence protocol messages and the corresponding state transitions.

Based on the policy during a write operation on shared data, cache coherence protocols fall into two main categories, *write-update* based protocols and *write-invalidate* based protocols. Write-update protocols involve interprocessor communication on every write operation to shared data. Write-invalidate protocols maintain cache coherence by invalidating copies of a memory block when the block is modified by a processor. The advantage of this is that at most the first write, in a sequence of writes to the same block with no intervening read operations from other processors, causes inter-processor communication. Hence, subsequent write operations can be completed locally until the same data block is requested by another processor. As such, write-invalidate protocols yield less network traffic and accordingly provide a better scalable solution to the cache coherence problem but they suffer from the high read latency induced by cache coherence misses. On the other hand, previous research has shown that coherence misses or cache-to-cache misses comprise a big portion of overall cache misses in shared memory systems and write-update protocols can reduce these coherence misses . However, write-update protocols tend to generate high traffic on the interconnect due to unnecessary updates especially for migratory objects. So, a single cache coherence policy solution to effectively provide performance for different types of

applications is not feasible. Having said that, it is difficult to implement a write-update based protocol on unordered interconnects and in general performance improvements from the protocol are offset by having a sequentially consistent memory system for these protocols. Therefore, most contemporary systems implement a write-invalidate based cache coherence protocol. However, an efficient adaptive/hybrid protocol to combine the benefits of both schemes can be an asset for future CMPs. Thereby; in this thesis, we developed and presented various adaptive cache coherence mechanisms that are optimized based on the CMP specific architectural characteristics.

7.2 Summary of Contributions

The main contributions presented in this dissertation fall in the following areas:

Empirical study for evaluating and exploring CMPs:

In order to achieve the research objectives, a top-down approach is followed starting with a comprehensive performance analysis study of CMPs. Major performance issues are identified and the performance impact of cache coherence overhead is evaluated for CMP based systems.

It was demonstrated that the cache coherence protocol performance can have severe effects for a wide range of platforms. It was also shown that the effects become even larger with the increasing number of cores. Accordingly, future CMPs must deploy efficient cache coherence mechanisms to be able to harness the potential performance benefits. As such, it was noted that there is a need for the design and exploration of cache coherence protocols that are targeting CMPs. Furthermore, it was demonstrated that symmetry of network on-chip is important, and can become an issue unless intelligently designed. Therefore, our

proposed cache coherence schemes were designed with these observations in mind. As such, they can adaptively work to improve the performance in CMP based architectures.

Producer-Consumer Adaptive Multicore (PCAM) Cache Coherence Protocol:

Cache coherence has been investigated heavily in the research community, but most of the prior work targeted multiprocessor machines (MPs) comprised of multiple single-core processors. With the advent of CMPs, cache coherence becomes a first-order design issue at the chip level. Many constraints including interconnect, latency, power, and energy differ in MP and CMP designs. In addition, most contemporary cache coherence protocols use a write-invalidate policy which tends to generate high coherence misses. This can become a major performance bottleneck as the number of cores inside a single chip increases.

As a solution to alleviate the coherence misses' impact on performance, we proposed to extend the existing directory controllers inside the chip to dynamically detect data sharing patterns of running workloads. Current optimizations are targeting the detection of producer-consumer sharing patterns. For such prediction, we proposed a specialized cache called Producer-Consumer Predictor Cache (PCPC). This cache holds the directory information as well as extra bits to keep track of the history of accessed cache lines that are homed by that specific core. PCPC includes a *recent_sharers* field to designate the potential consumers. It uses counters to decide whether the consumer set is stable and also to train down the predictor when necessary to avoid unnecessary updates. As such, PCPC predicts potential consumers for a cache line and helps to decide when to trigger speculative updates to these consumers. In addition, this prediction mechanism needs to be tightly coupled with the cache coherence protocol. Thus, we proposed to extend an existing write-invalidate protocol that can trigger write-update optimizations based on the feedback from PCPC. The resulting cache coherence protocol, *Producer-Consumer Adaptive Multicore*

(PCAM) protocol, works as an adaptive cache coherence protocol based on the data sharing pattern of the application dynamically by utilizing the feedback from the prediction mechanism. The goal was to have a simple approach with hardware requirements to avoid large design modifications. As such, the proposed mechanisms do not change the memory model seen by the programmer and compiler. The proposed adaptive protocols can be implemented on top of any write-invalidate protocol and can work with existing language, compiler, and processor designs. Furthermore, this adaptive protocol improved upon existing adaptive cache coherence protocols with the following details:

- Costly WRITEBACKs to main memory during consumer updates were eliminated to stay on-chip as much as possible for communication.
- A novel dynamic trigger mechanism is introduced to establish a data-forwarding protocol to be able to update potential consumers.

Extensive analytical as well as experimental evaluations, conducted on a tiled-CMP via full-system simulator, were used to assess the performance gains from our proposed dynamic hybrid protocols. Performance analysis is presented on a set of scientific applications from the SPLASH-2 and the NAS parallel benchmark suites. Results showed that the proposed mechanisms reduce cache-to-cache misses up to 48% which in turn improves application performance by up to 25%. Although performance speedup numbers are highly sensitive to the implementation and underlying toolset restrictions, we show them to represent the overall effect on the execution time. We believe further performance gains can be obtained by more aggressive prediction mechanisms.

Bandwidth Adaptive Mechanism

The designed adaptive cache coherence protocol was enhanced with a bandwidth adaptive

mechanism, based on the dynamic utilization of the underlying network. This mechanism was integrated into the PCAM protocol resulting in the "*Producer-Consumer Bandwidth Adaptive Multicore (PCBAM)*" protocol. PCBAM was proven to perform well under varying interconnect utilizations. For low-bandwidth scenarios, our proposed protocol prevented any performance loss that might happen with the write-update optimizations. In addition, when bandwidth is plentiful our proposed solution outperforms the base directory protocol. To the best of our knowledge, PCBAM is the first cache coherence protocol that utilizes a bandwidth-adaptive mechanism to implement a hybrid protocol that switches between write-invalidate and write-update policies.

Proximity Aware Mechanism

We also extended the PCAM protocol with a proximity aware mechanism, resulting in the "*Producer-Consumer Proximity Adaptive Multicore (PCPAM)*" protocol. The proximity-aware mechanism aims to establish latency based updates for consumers. It was designed based on the assumption that updates to far remote caches can be costly in terms of network traffic and accordingly may degrade performance. PCPAM pushes updates to consumers only within certain proximity. To the best of our knowledge, PCPAM is the first cache coherence protocol that utilizes a proximity-aware mechanism to implement a hybrid protocol that switches between write-invalidate and write-update policies.

Analytical Model

Due to several orders of magnitude slowdowns in full-system simulation studies, an analytical model was also considered as an alternative. Thereby, we adapted a high-level performance model from an earlier work by Srbljic et al. [134] to assess the potential gains from such dynamic adaptations and provide a fast tool for design space explorations when

the core count is high.

7.3 Future Work

As it is discussed in this thesis, on-chip communication is crucial to get better performance from CMPs. Heterogeneous interconnects can be deployed for efficient utilization of on-chip interconnection resources. Accordingly, on-chip wires can be designed to have different latency, bandwidth, and energy characteristics. Similarly, our proposed adaptive protocols can be extended to divert the data update traffic to high-bandwidth low-latency wires. In the meantime, normal coherence messaging traffic can stay on energy efficient links to reduce power consumption. In addition, our bandwidth adaptive mechanism can be extended to exploit such heterogeneous interconnects.

Larger shared memory systems can be built by using CMPs as building blocks. Such systems will have multiple layers of hierarchy to access the globally addressable shared memory. In addition, future many-core chips can utilize hierarchical policies to improve cache coherence performance. As such, our design mechanisms and proposed protocols can be used as building blocks of a hierarchical cache coherence protocol in order to accommodate the emerging many-core systems and multi-CMP platforms. Furthermore, our proximity-aware mechanism can be extended to map the write-update optimizations into the ununderlying hardware based on this hierarchy information.

Our proposed protocols rely on the prediction mechanism to trigger write-update optimizations. As such, more aggressive prediction mechanisms can be tested for potential performance benefits. Some synthetic micro-benchmarks can be developed to test corner cases and accordingly they can be used to characterize the performance impact of the prediction mechanism for various data sharing scenarios. Moreover, dynamic mechanisms can be

tested to establish stable producer-consumer sets depending on the application. Also, more enhanced down-training mechanisms can be investigated to eliminate any potential false positives from the prediction mechanism. To balance performance and area budget, the analytical model can be extended to include power and area. This will enable the inclusion of area and power as design constraints.

Modern programming languages typically follow the Partitioned Global Address Space (PGAS) model. Under this model, compilers can be aware of which data is shared and as well as the affinity of such data to cores/nodes. This can provide some compile-time useful information that can be used for selective updates in a similar scheme to what we have proposed in this thesis. Exploiting such compile-time information under PGAS languages for cache coherence using selective updates can make a very rich research area for future investigations.

Bibliography

[1] S. R. Alam, R. F. Barrett, J. A. Kuehn, P. C. Roth, and J. S. Vetter, "Characterization of scientific workloads on systems with multi-core processors.," in *IISWC*, pp. 225–236, IEEE, 2006.

[2] S. Balakrishnan, R. Rajwar, M. Upton, and K. Lai, "The impact of performance asymmetry in emerging multicore architectures," in *ISCA '05: Proceedings of the 32nd annual International Symposium on Computer Architecture*, (Washington, DC, USA), pp. 506–517, IEEE Computer Society, 2005.

[3] M. Chu, R. Ravindran, and S. Mahlke, "Data Access Partitioning for Fine-grain Parallelism on Multicore Architectures," in *MICRO '07: Proceedings of the 40th Annual IEEE/ACM International Symposium on Microarchitecture*, (Washington, DC, USA), pp. 369–380, IEEE Computer Society, 2007.

[4] P. F. Gorder, "Multicore processors for science and engineering," *Computing in Science and Eng.*, vol. 9, no. 2, pp. 3–7, 2007.

[5] G. E. Moore, "Cramming more components onto integrated circuits," *Electronics*, vol. 38, pp. 114–117, April 1965.

[6] D. Geer, "Industry trends: Chip makers turn to multicore processors.," *IEEE Computer*, vol. 38, no. 5, pp. 11–13, 2005.

[7] A. Agarwal and M. Levy, "The kill rule for multicore.," in *DAC '07: Proceedings of the 44th annual conference on Design automation*, pp. 750–753, IEEE, 2007.

[8] K. Olukotun and L. Hammond, "The future of microprocessors," *Queue*, vol. 3, no. 7, pp. 26–29, 2005.

[9] V. Agarwal, M. S. Hrishikesh, S. W. Keckler, and D. Burger, "Clock rate versus IPC: The end of the road for conventional microarchitectures.," in *ISCA '00: Proceedings of the 27th annual International Symposium on Computer architecture*, pp. 248–259, 2000.

[10] "Teraflops research chip, `http://techresearch.intel.com/articles/Tera-Scale/1449.htm`," June 2010.

[11] "Tile-gx100, a 100-core microprocessor from Tilera Corporation, `http://www.tilera.com`," May 2011.

[12] J. L. Hennessy and D. A. Patterson, *Computer Architecture, Fourth Edition: A Quantitative Approach*. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 2006.

[13] M. A. Heinrich, *The Performance and Scalability of Distribued Shared Memory Cache Coherence Protocols*. PhD thesis, Stanford University, 2008.

[14] M. M. K. Martin, D. J. Sorin, M. D. Hill, and D. A. Wood, "Bandwidth adaptive snooping.," in *International Symposium on High-Performance Computer Architecture (HPCA)*, pp. 251–262, 2002.

[15] N. Agarwal, L.-S. Peh, and N. K. Jha, "In-network snoop ordering (inso): Snoopy coherence on unordered interconnects.," in *HPCA*, pp. 67–78, IEEE Computer Society, 2009.

[16] M. Acacio, J. González, J. García, and J. Duato, "Owner prediction for accelerating cache-to-cache transfer misses in a cc-NUMA architecture," in *Proceedings of the 2002 ACM/IEEE conference on Supercomputing*, pp. 1–12, IEEE Computer Society Press Los Alamitos, CA, USA, 2002.

[17] P. Sweazey and A. J. Smith, "A class of compatible cache consistency protocols and their support by the ieee futurebus.," in *ISCA*, pp. 414–423, 1986.

[18] "AMD64 Architecture Programming Manual Volume 2: System Programming, `http://support.amd.com/us/Processor_TechDocs/24593.pdf`," June 2010.

[19] L. Cheng, *Context-Aware Coherence Protocols for Future Processors*. PhD thesis, University of Utah, 2007.

[20] H. Sutter, "The free lunch is over: A fundamental turn toward concurrency in software," *Dr. Dobb's Journal*, vol. 30, no. 3, pp. 202–210, 2005.

[21] M. R. Marty, *Cache Coherence Techniques For Multicore Processors*. PhD thesis, University of Wisconsin-Madison, 2008.

[22] A. Kayi, E. Kornkven, T. El-Ghazawi, S. Al-Bahra, and G. B. Newby, "Performance analysis and tuning for clusters with ccNUMA nodes for scientific computing - a case study," *International Journal of Computer Systems Science and Engineering*, vol. 24, September 2009.

[23] K. D. Bosschere, W. Luk, X. Martorell, N. Navarro, M. F. P. O'Boyle, D. N. Pnevmatikatos, A. Ramírez, P. Sainrat, A. Seznec, P. Stenström, and O. Temam, "High-performance embedded architecture and compilation roadmap.," *T. HiPEAC*, vol. 1, pp. 5–29, 2007.

[24] J. A. Brown, R. Kumar, and D. M. Tullsen, "Proximity-aware directory-based coherence for multi-core processor architectures.," in *SPAA '07: Proceedings of the 19th ACM Symposium on Parallelism in Algorithms and Architectures* (P. B. Gibbons and C. Scheideler, eds.), pp. 126–134, ACM, 2007.

[25] 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," Tech. Rep. UCB/EECS-2006-183, EECS Department, University of California, Berkeley, Dec 2006.

[26] P. Kundu and L.-S. Peh, "Guest editors' introduction: On-chip interconnects for multicores," *IEEE Micro*, vol. 27, no. 5, pp. 3–5, 2007.

[27] P. S. Magnusson, M. Christensson, J. Eskilson, D. Forsgren, G. Hållberg, J. Högberg, F. Larsson, A. Moestedt, and B. Werner, "Simics: A full system simulation platform.," *IEEE Computer*, vol. 35, no. 2, pp. 50–58, 2002.

[28] M. M. K. Martin, "Formal verification and its impact on the snooping versus directory protocol debate.," in *ICCD 2005: International Conference on Computer Design*, pp. 543–449, IEEE Computer Society, 2005.

[29] M. Acacio, J. Gonzalez, J. Garcia, and J. Duato, "A novel approach to reduce L2 miss latency in shared-memory multiprocessors," in *IPDPS '02: Proceedings of the International Parallel and Distributed Processing Symposium*, pp. 62–69, 2002.

[30] N. Eisley, L.-S. Peh, and L. Shang, "In-network cache coherence," in *MICRO 39: Proceedings of the 39th Annual IEEE/ACM International Symposium on Microarchitecture*, (Washington, DC, USA), pp. 321–332, IEEE Computer Society, 2006.

[31] L. Cheng, N. Muralimanohar, K. Ramani, R. Balasubramonian, and J. B. Carter, "Interconnect-aware coherence protocols for chip multiprocessors," in *ISCA '06:*

Proceedings of the 33rd annual International Symposium on Computer Architecture, (Washington, DC, USA), pp. 339–351, IEEE Computer Society, 2006.

[32] M. R. Marty, J. D. Bingham, M. D. Hill, A. J. Hu, M. M. K. Martin, and D. A. Wood, "Improving multiple-cmp systems using token coherence.," in *ISCA '05: Proceedings of the 32nd annual International Symposium on Computer Architecture*, pp. 328–339, IEEE Computer Society, 2005.

[33] A. Raynaud, Z. Zhang, and J. Torrellas, "Distance-adaptive update protocols for scalable shared-memory multiprocessors," in *HPCA '96: Proceedings of the Second International Symposium on High-Performance Computer Architecture*, pp. 323–334, Feb 1996.

[34] H. K. Grahn and P. Stenström, "Evaluation of a competitive-update cache coherence protocol with migratory data detection," *Journal of Parallel and Distributed Computing*, vol. 39, pp. 39–2, 1996.

[35] L. Cheng, J. B. Carter, and D. Dai, "An adaptive cache coherence protocol optimized for producer-consumer sharing.," in *International Symposium on High-Performance Computer Architecture (HPCA)*, pp. 328–339, IEEE Computer Society, 2007.

[36] M. M. K. Martin, M. D. Hill, and D. A. Wood, "Token coherence: decoupling performance and correctness," in *ISCA '03: Proceedings of the 30th annual International Symposium on Computer architecture*, (New York, NY, USA), pp. 182–193, ACM, 2003.

[37] M. M. K. Martin, *Token Coherence*. PhD thesis, University of Wisconsin-Madison, 2003.

[38] A. Raghavan, C. Blundell, and M. M. K. Martin, "Token tenure: Patching token counting using directory-based cache coherence.," in *MICRO*, pp. 47–58, IEEE Computer Society, 2008.

[39] A. Singhal, D. Broniarczyk, F. Cerauskis, J. Price, L. Yuan, C. Cheng, D. Doblar, S. Fosth, N. Agarwal, K. Harvey, *et al.*, "Gigaplane: A High Performance Bus for Large SMPs, Hot Interconnects IV," 1996.

[40] A. Charlesworth, S. Microsyst, and O. Beaverton, "Starfire: extending the SMP envelope," *Micro, IEEE*, vol. 18, no. 1, pp. 39–49, 1998.

[41] A. Moshovos, G. Memik, B. Falsafi, and A. Choudhary, "JETTY: filtering snoops for reduced energy consumption in SMP servers," in *HPCA '01: Proceedings of the Seventh International Symposium on High-Performance Computer Architecture*, pp. 85–96, 2001.

[42] A. Moshovos, "RegionScout: exploiting coarse grain sharing in snoop-based coherence," in *ISCA '05: Proceedings of the 32nd annual International Symposium on Computer Architecture*, (Washington, DC, USA), pp. 234–245, IEEE Computer Society, 2005.

[43] C.-M. Chung, J. Kim, and D. Kim, "Reducing snoop-energy in shared bus-based mpsocs by filtering useless broadcasts," in *GLSVLSI '07: Proceedings of the 17th ACM Great Lakes Symposium on VLSI*, (New York, NY, USA), pp. 126–131, ACM, 2007.

[44] A. Patel and K. Ghose, "Energy-efficient MESI cache coherence with pro-active snoop filtering for multicore microprocessors," in *ISLPED '08: Proceeding of the thirteenth International Symposium on Low power electronics and design*, (New York, NY, USA), pp. 247–252, ACM, 2008.

[45] N. Agarwal, L.-S. Peh, and N. K. Jha, "In-network coherence filtering: snoopy coherence without broadcasts.," in *MICRO* (D. H. Albonesi, M. Martonosi, D. I. August, and J. F. Martínez, eds.), pp. 232–243, ACM, 2009.

[46] V. Salapura, M. A. Blumrich, and A. Gara, "Design and implementation of the blue gene/p snoop filter.," in *HPCA*, pp. 5–14, IEEE Computer Society, 2008.

[47] M. Azimi, F. A. Briggs, M. Cekleov, M. Khare, A. Kumar, and L. P. Looi, "Scalability port: A coherent interface for shared memory multiprocessors.," in *Hot Interconnects*, pp. 65–70, IEEE Computer Society, 2002.

[48] W. Dally and B. Towles, "Route packets, not wires: on-chip interconnection networks," in *Design Automation Conference, 2001. Proceedings*, pp. 684 – 689, 2001.

[49] L. A. Barroso and M. Dubois, "Cache coherence on a slotted ring," in *International Conference on Parallel Processing (ICPP)*, pp. 230–237, 1991.

[50] Luiz André Barroso and Michel Dubois, "The performance of cache-coherent ring-based multiprocessors," in *ISCA '93: Proceedings of the 20th annual International Symposium on Computer architecture*, (New York, NY, USA), pp. 268–277, ACM, 1993.

[51] J. M. Tendler, J. S. Dodson, J. S. F. Jr., H. Le, and B. Sinharoy, "Power4 system microarchitecture.," *IBM Journal of Research and Development*, vol. 46, no. 1, pp. 5–26, 2002.

[52] B. Sinharoy, R. N. Kalla, J. M. Tendler, R. J. Eickemeyer, and J. B. Joyner, "Power5 system microarchitecture.," *IBM Journal of Research and Development*, vol. 49, no. 4-5, pp. 505–522, 2005.

[53] M. Kistler, M. Perrone, and F. Petrini, "Cell multiprocessor communication network: Built for speed," *Micro, IEEE*, vol. 26, pp. 10–23, May-June 2006.

[54] M. R. Marty and M. D. Hill, "Coherence ordering for ring-based chip multiprocessors.," in *MICRO*, pp. 309–320, IEEE Computer Society, 2006.

[55] K. Strauss, X. Shen, and J. Torrellas, "Flexible snooping: Adaptive forwarding and filtering of snoops in embedded-ring multiprocessors.," in *ISCA '06: Proceedings of the 33rd annual International Symposium on Computer Architecture*, pp. 327–338, IEEE Computer Society, 2006.

[56] E. E. Bilir, R. M. Dickson, Y. Hu, M. Plakal, D. J. Sorin, M. D. Hill, and D. A. Wood, "Multicast snooping: A new coherence method using a multicast address network.," in *ISCA '99: Proceedings of the 26th annual International Symposium on Computer architecture*, pp. 294–304, 1999.

[57] M. M. K. Martin, D. J. Sorin, A. Ailamaki, A. R. Alameldeen, R. M. Dickson, C. J. Mauer, K. E. Moore, M. Plakal, M. D. Hill, and D. A. Wood, "Timestamp snooping: an approach for extending smps.," in *ASPLOS IX: Proceedings of the 9th International conference on Architectural Support for Programming Languages and Operating Systems*, pp. 25–36, 2000.

[58] B. Cuesta, A. Robles, and J. Duato, "An effective starvation avoidance mechanism to enhance the token coherence protocol.," in *15th Euromicro International Conference on Parallel, Distributed and Network-Based Processing (PDP 2007)*, pp. 47–54, IEEE Computer Society, 2007.

[59] B. Cuesta, A. Robles, and J. Duato, "Improving token coherence by multicast coherence messages.," in *16th Euromicro International Conference on Parallel, Distributed and Network-Based Processing (PDP 2008), 13-15 February 2008, Toulouse, France*, pp. 269–273, IEEE Computer Society, 2008.

[60] B. Cuesta, A. Robles, and J. Duato, "Switch-based packing technique for improving token coherence scalability.," in *PDCAT*, pp. 83–90, IEEE Computer Society, 2008.

[61] D. Lenoski, J. Laudon, K. Gharachorloo, W.-D. Weber, A. Gupta, J. L. Hennessy, M. Horowitz, and M. S. Lam, "The stanford dash multiprocessor.," *IEEE Computer*, vol. 25, no. 3, pp. 63–79, 1992.

[62] J. Laudon and D. Lenoski, "The SGI Origin: A ccNUMA Highly Scalable Server.," in *ISCA '97: Proceedings of the 24th annual International Symposium on Computer Architecture*, pp. 241–251, 1997.

[63] S. S. Mukherjee, P. J. Bannon, S. Lang, A. Spink, and D. Webb, "The alpha 21364 network architecture.," *IEEE Micro*, vol. 22, no. 1, pp. 26–35, 2002.

[64] K. Gharachorloo, M. Sharma, S. Steely, and S. V. Doren, "Architecture and design of AlphaServer GS320.," in *ASPLOS IX: Proceedings of the 9th International conference on Architectural Support for Programming Languages and Operating Systems*, pp. 13–24, 2000.

[65] J. R. Feehrer, P. Rotker, M. Shih, P. Gingras, P. Yakutis, S. Phillips, J. Heath, and S. Turullols, "Coherency hub design for multi-node victoria falls server systems.," in *Hot Interconnects*, pp. 43–50, IEEE Computer Society, 2008.

[66] J. R. Feehrer, P. Rotker, M. Shih, P. Gingras, P. Yakutis, S. Phillips, and J. Heath, "Coherency hub design for multisocket sun servers with coolthreads technology.," *IEEE Micro*, vol. 29, no. 4, pp. 36–47, 2009.

[67] A. R. Karlin, M. S. Manasse, L. Rudolph, and D. D. Sleator, "Competitive snoopy caching.," *Algorithmica*, vol. 3, pp. 77–119, 1988.

[68] A. L. Cox and R. J. Fowler, "Adaptive cache coherency for detecting migratory shared data.," in *International Symposium on Computer Architecture (ISCA)*, pp. 98–108, 1993.

[69] P. Stenström, M. Brorsson, and L. Sandberg, "An adaptive cache coherence protocol optimized for migratory sharing.," in *International Symposium on Computer Architecture (ISCA)*, pp. 109–118, 1993.

[70] H. Nilsson and P. Stenström, "An adaptive update-based cache coherence protocol for reduction of miss rate and traffic," in *Proc. Parallel Architectures and Languages Europe (PARLE) Conf., Athens, Greece (Lecture Notes in Computer Science, 817*, pp. 363–374, Springer-Verlag, 1994.

[71] F. Dahlgren and P. Stenström, "Reducing the write traffic for a hybrid cache protocol.," in *International Conference on Parallel Processing (ICPP)*, pp. 166–173, 1994.

[72] F. Dahlgren, "Boosting the performance of hybrid snooping cache protocols," in *ISCA '95: Proceedings of the 22nd annual International Symposium on Computer architecture*, (New York, NY, USA), pp. 60–69, ACM, 1995.

[73] C. Anderson and A. R. Karlin, "Two adaptive hybrid cache coherency protocols.," in *International Symposium on High-Performance Computer Architecture (HPCA)*, pp. 303–313, 1996.

[74] J. Nilsson and F. Dahlgren, "Reducing ownership overhead for Load-Store sequences in cache-coherent multiprocessors.," in *IPDPS '00: Proceedings of the International Parallel and Distributed Processing Symposium*, pp. 684–692, IEEE Computer Society, 2000.

[75] S. J. Eggers and R. H. Katz, "Evaluating the performance of four snooping cache coherency protocols," *SIGARCH Comput. Archit. News*, vol. 17, no. 3, pp. 2–15, 1989.

[76] Susan J. Eggers and Randy H. Katz, "A characterization of sharing in parallel programs and its application to coherency protocol evaluation.," in *International Symposium on Computer Architecture (ISCA)*, pp. 373–382, 1988.

[77] J. Nilsson and F. Dahlgren, "Improving performance of load-store sequences for transaction processing workloads on multiprocessors.," in *International Conference on Parallel Processing (ICPP)*, pp. 246–, 1999.

[78] D. Gustavson, "The scalable coherent interface and related standards projects," *Micro, IEEE*, vol. 12, pp. 10–22, Feb 1992.

[79] L. Cheng and J. B. Carter, "Extending cc-numa systems to support write update optimizations.," in *SC '08: Proceedings of the 2008 ACM/IEEE conference on Supercomputing*, p. 30, IEEE/ACM, 2008.

[80] S. S. Mukherjee and M. D. Hill, "Using prediction to accelerate coherence protocols.," in *International Symposium on Computer Architecture (ISCA)*, pp. 179–190, 1998.

[81] T.-Y. Yeh and Y. N. Patt, "Alternative implementations of two-level adaptive branch prediction.," in *International Symposium on Computer Architecture (ISCA)*, pp. 124–134, 1992.

[82] S. Kaxiras and J. R. Goodman, "Improving cc-NUMA performance using instruction-based prediction.," in *International Symposium on High-Performance Computer Architecture (HPCA)*, pp. 161–, 1999.

[83] S. Kaxiras and C. Young, "Coherence communication prediction in shared-memory multiprocessors.," in *International Symposium on High-Performance Computer Architecture (HPCA)*, pp. 156–167, 2000.

[84] A.-C. Lai and B. Falsafi, "Memory sharing predictor: The key to a speculative coherent dsm.," in *ISCA '99: Proceedings of the 26th annual International Symposium on Computer architecture*, pp. 172–183, 1999.

[85] M. E. Acacio, J. González, J. M. García, and J. Duato, "The use of prediction for accelerating upgrade misses in cc-NUMA multiprocessors.," in *PACT '02: Proceedings of the 11th International conference on Parallel architectures and compilation techniques*, pp. 155–164, IEEE Computer Society, 2002.

[86] J. Nilsson, A. Landin, and P. Stenström, "The coherence predictor cache: A resource-efficient and accurate coherence prediction infrastructure.," in *IPDPS '03: Proceedings of the International Parallel and Distributed Processing Symposium*, p. 10, IEEE Computer Society, 2003.

[87] M. M. K. Martin, P. J. Harper, D. J. Sorin, M. D. Hill, and D. A. Wood, "Using destination-set prediction to improve the latency/bandwidth tradeoff in shared-memory multiprocessors.," in *International Symposium on Computer Architecture (ISCA)*, pp. 206–217, IEEE Computer Society, 2003.

[88] S. Leventhal and M. Franklin, "Perceptron based consumer prediction in shared-memory multiprocessors," in *ICCD 2006: International Conference on Computer Design*, pp. 148–154, Oct. 2006.

[89] D. Ghosh, J. B. Carter, and H. D. III, "Perceptron-based coherence predictors," in *Proc. of 2nd Workshop on Chip Multiprocessor Memory Systems and Interconnects (CMP-MSI), in conjunction with ISCA 2008*, 2008.

[90] D. Koufaty, X. Chen, D. K. Poulsen, and J. Torrellas, "Data forwarding in scalable shared-memory multiprocessors.," in *International Conference on Supercomputing (ICS)*, pp. 255–264, 1995.

[91] H. Abdel-Shafi, J. Hall, S. V. Adve, and V. S. Adve, "An evaluation of fine-grain producer-initiated communication in cache-coherent multiprocessors.," in *International Symposium on High-Performance Computer Architecture (HPCA)*, pp. 204–, 1997.

[92] G. Byrd and M. Flynn, "Producer-consumer communication in distributed shared memory multiprocessors," *Proceedings of the IEEE*, vol. 87, pp. 456–466, Mar 1999.

[93] E. Atoofian and A. Baniasadi, "A power-aware prediction-based cache coherence protocol for chip multiprocessors.," in *IPDPS '07: Proceedings of the International Parallel and Distributed Processing Symposium*, pp. 1–8, IEEE, 2007.

[94] A. Ros, M. E. Acacio, and J. M. García, "Dico-cmp: Efficient cache coherency in tiled cmp architectures.," in *IPDPS 08': 22nd IEEE International Symposium on Parallel and Distributed Processing, Miami, Florida USA, April 14-18, 2008*, pp. 1–11, IEEE, 2008.

[95] A. Kayi, Y. Yao, T. A. El-Ghazawi, and G. Newby, "Experimental evaluation of emerging multi-core architectures.," in *IPDPS '07: Proceedings of the International Parallel and Distributed Processing Symposium; March 26-30 2007; Long Beach, CA, USA*, pp. 1–6, IEEE, 2007.

[96] A. Kayi, T. El-Ghazawi, and G. B. Newby, "Performance issues in emerging homogeneous multi-core architectures," *Simulation Modelling Practice and Theory*, vol. 17, pp. 1485–1499, Oct. 2009.

[97] "High Performance Computing Modernization Program technology insertion home page, `http://www.hpcmo.hpc.mil/bizopps/TI/index.php`," February 2009.

[98] "HPC Challenge Benchmarking Suite, `http://icl.cs.utk.edu/hpcc`," May 2011.

[99] L. W. McVoy and C. Staelin, "lmbench: Portable tools for performance analysis.," in *USENIX Annual Technical Conference*, pp. 279–294, 1996.

[100] "PAPI home page, `http://icl.cs.utk.edu/papi`," July 2008.

[101] J. McCalpin, "Memory bandwidth and machine balance in current high performance computers," *IEEE Technical Committee on Computer Architecture Newsletter*, pp. 19–25, 1995.

[102] D. Pase and J. Stephens, "Performance of Two-Way Opteron and Xeon Processor-Based Servers for Scientific and Technical Applications," in *Linux Clusters: The HPC Revolution*, 2005.

[103] "Effective Bandwidth benchmark home page, `http://www.hlrs.de/organization/par/services/models/mpi/b_eff`," July 2007.

[104] "LAMMPS home page, `http://lammps.sandia.gov/`," July 2007.

[105] "HYCOM model home page, `http://oceanmodeling.rsmas.miami.edu/hycom`," July 2007.

[106] "NAMD scalable molecular dynamics home page, `http://www.ks.uiuc.edu/Research/namd`," July 2007.

[107] "Weather Research Forecasting (WRF) model home page, `http://www.wrf-model.org`," July 2007.

[108] C. A. Rendleman, V. E. Beckner, M. Lijewski, W. Crutchfield, and J. B. Bell, "Parallelization of structured, hierarchical adaptive mesh refinement algorithms," *Computing and Visualization in Science*, vol. 3, no. 3, pp. 147–157, 2000.

[109] "ERDC major shared resource center, special edition: HPCMP Benchmarking, http://www.erdc.hpc.mil/PDF/spring05.pdf," January 2008.

[110] J. McGlaun, S. Thompson, and M. Elrick, "Cth: A three-dimensional shock wave physics code," *International Journal of Impact Engineering*, vol. 10, no. 1-4, pp. 351 – 360, 1990.

[111] S. Saini, D. Talcott, D. Jespersen, M. J. Djomehri, H. Jin, and R. Biswas, "Scientific application-based performance comparison of SGI Altix 4700, IBM POWER5+, and SGI ICE 8200 supercomputers.," in *SC*, p. 7, IEEE/ACM, 2008.

[112] "ChaNGa home page, `http://www-pcc.astro.washington.edu/tools/changa.html`," July 2007.

[113] X. Qin and J.-L. Baer, "Optimizing software cache-coherent cluster architectures," in *Proceedings of the 1998 ACM/IEEE conference on Supercomputing (CDROM)*, Supercomputing '98, (Washington, DC, USA), pp. 1–14, IEEE Computer Society, 1998.

[114] P. Mardahl, A. Greenwood, T. Murphy, and K. Cartwright, "Parallel performance characteristics of icepic," *Defense Users Group Conference, Department of*, vol. 0, p. 86, 2003.

[115] "High Performance Computing Modernization Program home page, `http://www.hpcmo.hpc.mil/index.html`," February 2009.

[116] "Charm++ home page, `http://charm.cs.uiuc.edu`," July 2008.

[117] S. C. Woo, M. Ohara, E. Torrie, J. P. Singh, and A. Gupta, "The SPLASH-2 Programs: Characterization and methodological considerations.," in *ISCA '95: Proceedings of the 22nd annual International Symposium on Computer architecture*, pp. 24–36, 1995.

[118] A. R. Alameldeen, M. M. K. Martin, C. J. Mauer, K. E. Moore, M. Xu, M. D. Hill, D. A. Wood, and D. J. Sorin, "Simulating a $2m commercial server on a $2k pc.," *IEEE Computer*, vol. 36, no. 2, pp. 50–57, 2003.

[119] "AMD Direct Connect Architecture, `http://www.amd.com/us/products/technologies/direct-connect-architecture/Pages/direct-connect-architecture.aspx`," May 2011.

[120] F. Delattre, "Intel Core 2 Duo - Test, `http://www.behardware.com/articles/623-1/intel-core-2-duo-test.html`," July 2006.

[121] A. Kayi, E. Kornkven, T. El-Ghazawi, S. Al-Bahra, and G. Newby, "Performance Evaluation of Clusters with ccNUMA Nodes-A Case Study," in *HPCC '08: Proceedings of the 10th IEEE International Conference on High Performance Computing and Communications; Sept. 25-27 2008; Dalian, China*, pp. 320–327, 2008.

[122] K. Datta, M. Murphy, V. Volkov, S. Williams, J. Carter, L. Oliker, D. Patterson, J. Shalf, and K. A. Yelick, "Stencil computation optimization and auto-tuning on state-of-the-art multicore architectures.," in *SC*, p. 4, IEEE/ACM, 2008.

[123] "Sun Fire X4600/X4600 M2 server system architecture technical white paper, `http://www.sun.com/servers/x64/x4600/arch-wp.pdf`," July 2007.

[124] A. Kayi, E. Kornkven, T. A. El-Ghazawi, and G. Newby, "Application performance tuning for clusters with ccnuma nodes.," in *CSE'08: Proceedings of the 11th IEEE International Conference on Computational Science and Engineering, , 16-18 July 2008, São Paulo, SP, Brazil*, pp. 245–252, IEEE Computer Society, 2008.

[125] 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 Comput. Archit. News*, vol. 33, no. 4, pp. 92–99, 2005.

[126] "Linux NUMA API, `http://www.novell.com/collateral/4621437/4621437.pdf`," July 2007.

[127] O. Serres, A. Anbar, S. G. Merchant, A. Kayi, and T. El-Ghazawi, "Address translation optimization for Unified Parallel C multi-dimensional arrays," in *2011 IEEE International Symposium on Parallel & Distributed Processing, Workshops and Phd Forum (IPDPSW)*, pp. 1–8, IEEE, 2011.

[128] C. Fensch and M. Cintra, "An OS-based alternative to full hardware coherence on tiled CMPs," in *HPCA '08: Proceedings of the IEEE 14th International Symposium on High Performance Computer Architecture*, pp. 355–366, Feb. 2008.

[129] 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*, vol. 27, pp. 15–31, Sept.-Oct. 2007.

[130] W. Lee, R. Barua, M. Frank, D. Srikrishna, J. Babb, V. Sarkar, and S. P. Amarasinghe, "Space-time scheduling of instruction-level parallelism on a raw machine.," in *ASPLOS*, pp. 46–57, 1998.

[131] K. Mai, T. Paaske, N. Jayasena, R. Ho, W. J. Dally, and M. Horowitz, "Smart memories: a modular reconfigurable architecture.," in *ISCA*, pp. 161–171, 2000.

[132] K. Sankaralingam, R. Nagarajan, H. Liu, C. Kim, J. Huh, N. Ranganathan, D. Burger, S. W. Keckler, R. G. McDonald, and C. R. Moore, "Trips: A polymorphous architecture for exploiting ilp, tlp, and dlp.," *TACO*, vol. 1, no. 1, pp. 62–93, 2004.

[133] S. Swanson, A. Schwerin, M. Mercaldi, A. Petersen, A. Putnam, K. Michelson, M. Oskin, and S. J. Eggers, "The wavescalar architecture.," *ACM Trans. Comput. Syst.*, vol. 25, no. 2, 2007.

[134] S. Srbljic, Z. Vranesic, M. Stumm, and L. Budin, "Analytical prediction of performance for cache coherence protocols," *Computers, IEEE Transactions on*, vol. 46, pp. 1155–1173, Nov 1997.

[135] M. Dubois and J.-C. Wang, "Shared block contention in a cache coherence protocol.," *IEEE Trans. Computers*, vol. 40, no. 5, pp. 640–644, 1991.

[136] "NAS Parallel Benchmarks, `http://www.nas.nasa.gov/Resources/Software/npb.html`," May 2011.

[137] "NAS Parallel Benchmarks, OpenMP version developed by omni group, `http://www.hpcs.cs.tsukuba.ac.jp/omni-openmp`," May 2009.