The Dissertation Committee for Jaehyuk Huh

certifies that this is the approved version of the following dissertation:

Hardware Techniques to Reduce Communication Costs in Multiprocessors

Committee:

Douglas C. Burger, Supervisor

Stephen W. Keckler

James C. Browne

Craig M. Chase

Calvin Lin

Hardware Techniques to Reduce Communication Costs in Multiprocessors

by

Jaehyuk Huh, B.S., M.S.

Dissertation

Presented to the Faculty of the Graduate School of

The University of Texas at Austin

in Partial Fulfillment

of the Requirements

for the Degree of

Doctor of Philosophy

The University of Texas at Austin

May 2006

Dedicated to my wife and my parents

Acknowledgments

I wish to thank the numerous people who helped me. I would like to thank my advisor, Dr. Douglas C. Burger, for the inspiration and guidance that he has given me. I would also like to thank my co-advisor, Dr. Stephen W. Keckler for his guidance. I would like to thank my fellow students in the Computer Architecture and Technology Laboratory for their input and discussions. I would like to thank Simha Sethumadhaven for his always generous encouragement and occasional sharp criticism. I would also like to thank Changku Kim, to whom I can speak my mother language, for our collaboration for the CMP NUCA study. Finally, I would like to thank my wife, Young-ri Choi. As wife, friend, and colleague, she has supported me more than anyone else. Without her, I could not have finished the rough road to this moment.

I thank IBM Corporation for my fellowship and the SimOS-PowerPC simulator. Funding for this research was provided in part by IBM Ph.D. Fellowships for the years 2004-2005, and in part by the Defense Advanced Research Projects Agency (DARPA) under contracts F33615-01-C-1892 and NBCH30390004.

Jaehyuk Huh

The University of Texas at Austin

May 2006

Hardware Techniques to Reduce Communication Costs in Multiprocessors

Publication No. _____

Jaehyuk Huh, Ph.D.

The University of Texas at Austin, 2006

Supervisor: Douglas C. Burger

This dissertation explores techniques for reducing the costs of inter-processor communication in shared memory multiprocessors (MP). We seek to improve MP performance by enhancing three aspects of multiprocessor cache designs: miss reduction, low communication latency, and high coherence bandwidth. In this dissertation, we propose three techniques to enhance the three factors: shared non-uniform cache architecture, coherence decoupling, and subspace snooping.

As a miss reduction technique, we investigate shared cache designs for future Chip-Multiprocessors (CMPs). Cache sharing can reduce cache misses by eliminating unnecessary data duplication and by reallocating the cache capacity dynamically. We propose a reconfigurable shared non-uniform cache architecture and evaluate the trade-offs of cache sharing with varied sharing degrees. Although shared caches can improve caching efficiency, the most significant disadvantage of shared caches

is the increase of cache hit latencies. To mitigate the effect of the long latencies, we evaluate two latency management techniques, dynamic block migration and L1 prefetching.

However, improving the caching efficiency does not reduce the cache misses induced by MP communication. For such communication misses, the latencies of cache coherence should be either reduced or hidden and the coherence bandwidth should scale with the number of processors. To mitigate long communication latencies, coherence decoupling uses speculation for communication data. Coherence decoupling allows processors to run speculatively at communication misses with predicted values. Our prediction mechanism, called Speculative Cache Lookup (SCL) protocol, uses stale values in the local caches. We show that the SCL read component can hide false sharing and silent store misses effectively. We also investigate the SCL update component to hide the latencies of truly shared misses by updating invalid blocks speculatively.

To improve the coherence bandwidth, we propose subspace snooping, which improves the snooping bandwidth with future large-scale shared-memory machines. Even with huge optical bus bandwidth, traditional snooping protocols may not scale to hundreds of processors, since all processors should respond to every bus access. Subspace snooping allows only a subset of processors to be snooped for a bus access, thus increasing the effective snoop tag bandwidth. We evaluate subspace snooping with a large broadcasting bandwidth provided by optical interconnects.

Table of Contents

List of Tables

List of Figures

Chapter 1

Introduction

In the past decade, multiprocessors have emerged from the parallel scientific computing to commercial server applications. With the introduction of recent chip multiprocessors, multiprocessors have moved into the personal computing domain and are starting to move into embedded systems, making multiprocessors ubiquitous in every area of computation. In multiprocessors, communication among processors has a significant effect on the system performance, and there have been numerous studies to improve multiprocessors for low communication overheads. However, there are several technology trends, making optimizing communication in multiprocessors more complicated:

- Increasing transistor density has made possible single chip-multiprocessors (CMPs). The on-chip integration of multiprocessors allows more flexible organization of caches and communication networks than chip-to-chip multiprocessors by eliminating the limit of off-chip bandwidth. As on-chip transistor counts are growing close to billions of transistors, tens of processing cores will

be placed on chips in the near future. Efficient organizations of on-chip communication mechanisms will be essential in the designs of high-performance CMPs.

- Performance losses due to communication among processors have been increasing. Although clock speed increase has recently slowed down due to power consumption limitation, inter-processor communication latencies are still larger than local cache hit latencies by one or two orders of magnitude. Particularly, in commercial workloads, the inter-processor communication latencies affect system performance significantly, due to fine-grained data sharing in the commercial servers. Barroso et al. have showed that about 50% of cache misses in server workloads are misses induced by inter-processor communication [7]. The increased performance losses from communication latencies have forced hardware designers to spend more time on optimizing interconnection network designs and coherence protocols.

- Interconnection network bandwidth has been increasing with better electrical signaling technologies. Furthermore, the advances in optical interconnection technologies can potentially provide enormous interconnection bandwidth, mitigating the communication bottlenecks of large scale multiprocessors. However, in shared memory multiprocessors, cache coherence mechanisms limit the communication bandwidth. It is necessary to improve other components of coherence systems, such as snoop tags in snooping protocols, with interconnection bandwidth.

1.1 Communication Costs in Shared Memory Multiprocessor

The memory hierarchies of shared-memory multiprocessors consist of multiple levels of local caches and the external main memory. The cache coherence mechanisms maintain the consistency of data among the local caches and memory. In the shared memory model, processors communicate with each other implicitly through the cache hierarchy and the coherence mechanism. Accesses to remote caches (the caches of other processors) and main memory through the coherence mechanism are expensive, since the latencies are much longer than local cache accesses and the bandwidth is limited by off-chip interconnections.

To improve the performance of MP systems, local caches should absorb as many accesses as possible to avoid the expensive accesses to the remote caches and memory. However, some remote accesses are unavoidable, and lead to significant performance losses in multiprocessors. For such remote accesses, both latency and bandwidth should be improved for better performance of MP systems. Therefore, techniques for reducing the communication overheads can be classified to three categories: miss reduction, low latency communication, and high communication bandwidth.

- Reducing cache misses (improving cache efficiency): The most efficient way of reducing the communication overheads is to reduce unnecessary remote accesses. Traditionally, increasing capacity and associativity has improved the caching efficiency. However, in CMPs with multiple processors on a chip, caching efficiency can be further improved by sharing cache capacity among

processors [80, 81]. Cache sharing can also improve communication among processors on a chip by providing a fast and high bandwidth coherence mechanism within a chip.

- Hiding communication latencies: True communication among processors can not be reduced by increasing caching efficiency. For such communication-induced misses, the latencies of acquiring updated values should be minimized. Traditional techniques to reduce the latencies are to shorten interconnection network latencies, and to optimize cache coherence mechanisms [18, 101, 21]. Alternatively, prefetching communication data and initiating coherence transactions speculatively hide the effect of long latencies [58, 79, 49, 56].

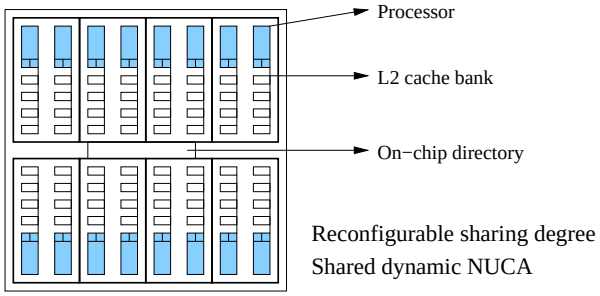
- Improving communication bandwidth: In shared-memory multiprocessors, the bandwidth of cache coherence mechanism determines communication bandwidth. Even with the advances in network bandwidth, increasing the coherence bandwidth grows complicated due to the protocol complexity and the energy consumption.

The next section introduces three techniques to enhance the three aspects of multiprocessor performance.

1.2 Three Techniques to Improve Multiprocessor Memory Systems

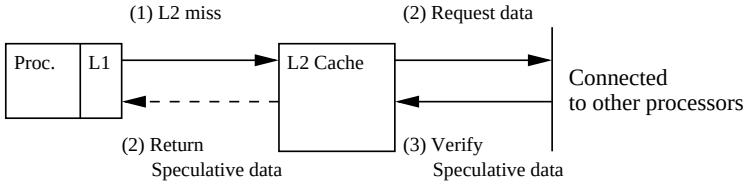
Figure 1.1 presents three techniques improving each aspect of multiprocessor memory systems: miss reduction, low communication latency, and high bandwidth. First,

1. Miss Reduction (C−NUCA): Shared caches for CMPs



---

2. Latency (Coherence Decoupling): Speculation to hide communication latency



---

3. Bandwidth (Subspace Snooping): Increasing snooping coherence bandwidth

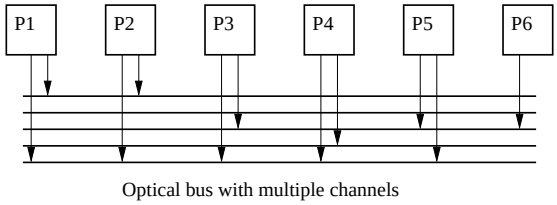


Figure 1.1: Improving MP memory systems: miss reduction, low latency and high bandwidth

to reduce off-chip accesses in CMPs, we propose a configurable shared cache design. However, the shared caches can have long hit latencies, reducing their benefits. To mitigate the hit latencies, our shared caches use the dynamic block migration technique.

Second, we use speculation to hide communication latencies. Current out-of-order processors can execute instructions speculatively and recover when the speculation is incorrect. *Coherence decoupling* predicts communication data for a coherence miss and allows the processor to run speculatively while the safe cache coherence protocol obtains the correct data. The speculation on communication data can hide long communication latencies without changing underlying coherence protocols.

Third, we propose a new class of coherence protocols, called *subspace snooping* to improve coherence bandwidth. The advances of interconnection technologies such as optical buses, have been increasing the bus bandwidth tremendously. However, traditional snooping protocols, despite of their advantages over directory protocols, can not scale due to the limited snoop tag bandwidth. The new protocol will improve the effective snoop tag bandwidth on optical buses.

1.2.1 Reducing Cache Misses: Shared Cache Organization for CMPs

Traditionally, cache misses can be reduced by increasing cache sizes (capacity misses) and associativity (conflict misses). However, the increases of capacity and associativity may increase the hit latencies of caches, reducing misses at the expense of the increased delays for hits. The optimal cache configuration balances the benefit of reduced misses and the performance loss due to the increased latencies.

Recent advances in CMP technologies have provided a new way to reduce off-chip misses by sharing caches. Unlike a single processor with a dedicated local cache, CMPs allow the tight integration of multiple processors and caches on a single chip. Since the connectivity among processors (on the same chip) is not limited by off-chip pin bandwidth, processors can share a large pool of on-chip cache capacity.

Cache sharing can reduce cache misses in several different ways. First, shared data do not need to be cached in multiple places, wasting cache capacity. In traditional private caches, shared blocks can be copied to multiple private caches. When a cache is shared by on-chip processors, the shared working set of the processors has only one copy of data in the shared cache. Such single-copy property can improve the caching efficiency, and thus reduce capacity misses. Second, cache capacity can be dynamically reallocated among processors. In private caches, only a dedicated processor can use the local cache capacity. However, when there is a large working set disparity, some private caches may have unused cache spaces, but others may suffer from capacity misses. Cache sharing allows processors to share capacity dynamically, improving caching efficiency by capacity rearrangement.

This dissertation evaluates the trade-offs of cache sharing in CMPs to reduce off-chip misses. Cache sharing can increase hit latencies, since cache sizes should be larger and provide higher bandwidth than private caches. Therefore, with many processors in future CMPs, the optimal number of processors sharing a cache is a first order design issue. We propose a CMP substrate which can reconfigure the number of sharing processors (or *sharing degree*). With the reconfigurable substrate, this dissertation evaluates CMP performance with varying sharing degrees.

The most significant disadvantage of shared caches is the increased hit la-

tencies. This dissertation investigates two techniques to reduce the effect of hit latencies in shared caches.

First, the reconfigurable CMP substrate supports dynamic block migration. Dynamic Non-Uniform Cache Architecture (D-NUCA) consists of multiple independent banks with network fabrics [53]. A cache block can be mapped in multiple banks, and access latencies from processors to banks vary by the distances. Blocks are moved from farther-located banks to closer-located banks, migrating frequently accessed blocks closer to a given processor. D-NUCA can reduce average hit latencies, since the majority of blocks are found in the banks close to the processors.

Second, L1 prefetching mechanisms can hide long L2 hit latencies. L1 prefetching does not reduce the L2 access latencies, but hides the latencies by moving data to the L1 caches before processors access them [6, 47]. This dissertation evaluates stride-based hardware prefetching engines for CMPs. The prefetching engines recognize regular strides between successively missed addresses, and send the request to the L2 shared cache as early as possible.

1.2.2 Hiding Communication Latency: Speculation for Cache Coherence

In shared memory multiprocessors, processors communicate with each other through cache coherence protocols. In traditional invalidation-based coherence protocols, any write to shared data forces shared copies to be invalidated, and subsequent reads to the invalid local copies will cause cache misses. Such misses are defined as *communication misses* or *coherence misses* [20]. Communication misses can not be reduced by increasing local cache sizes or associativity, since such misses are induced

by producer-consumer communication.

A straightforward way of improving multiprocessors for communication misses is to reduce the latency for obtaining data from the main memory or the other caches. Interconnection networks can be improved to shorten the latency and cache coherence protocols can also be optimized. In this dissertation, we propose a novel way of reducing the effect of long communication latencies, using speculation. Speculation is a technique to hide the effect of long latency events. The outcome of an event is predicted and validated later. Modern microprocessors have the capability to execute instructions speculatively.

We propose a technique called *coherence decoupling* to use speculation for coherence misses. For a coherence (communication) miss, the missed data are predicted, and the processor continues to execute dependent instructions. While the processor runs without stalls, a backing coherence protocol obtains non-speculative (correct) data by communicating with the other processors and the main memory. When the correct data return, the predicted value is validated against the correct value. If the prediction was incorrect, the processor state is rolled back to the last correct state before the speculation.

Coherence decoupling breaks a traditional cache coherence protocol into two parts, a Speculative Cache Lookup (SCL) protocol and a safe coherence protocol. The SCL protocol provides speculative load values for early use in communication misses. The backing verification protocol, which is much slower than the aggressive SCL protocol, guarantees the correctness of coherence. Using the SCL protocol, processing cores continue to execute instructions speculatively, while the safety net verifies the execution correctness. This speculative execution can hide the long

latency of coherence misses.

Coherence decoupling decouples the performance of cache coherence from the correctness. Improving the SCL protocols for better performance does not add complexity to the underlying cache coherence protocol. Even if the backing coherence protocol is slow, the fast value prediction for communication data can hide much of the latency. Coherence decoupling also reduces the application programmer's burden to fine-tune the programs, as application programmers spend more time to remove unnecessary communication in MP systems.

1.2.3 Increasing Communication Bandwidth: A New Snooping Method for High-Bandwidth SMPs

In the shared memory model with hardware cache coherence, the available communication bandwidth is dependent on the bandwidth of the coherence mechanism. The cache coherence mechanism transfers the requested data to the local caches as well as maintains the integrity of data.

Traditionally, there have been two mechanisms of hardware cache coherence, snooping protocols and directory protocols. Snooping protocols (or bus-based coherence) broadcast every memory request to all the lowest-level (inclusive) caches in the system. Since every cache can observe memory transactions, the data consistency is maintained by each cache in a distributed way. On the other hand, directory protocols use an external directory to keep track of current sharers of data. Memory requests are sent only to the sharing processors, reducing the bandwidth consumption of each request. The directory protocols may scale better than snooping protocols, at the expense of the increased complexity and 3-way communication

through directories.

Of the two types of coherence mechanisms, the snooping protocols have been more widely adopted for commercial multiprocessors. The bandwidth of snooping protocols has been improved by orders of magnitude, and the available bandwidth has been large enough to support commercial multiprocessors with tens of processors [38, 15]. The snooping protocols are less prone to protocol errors than the complex directory protocols. Fast cache-to-cache data transfers are possible in snooping protocols, since no directory lookup is necessary.

In current snooping protocols for commercial servers, the address bus and data bus are separated to increase the bandwidth. Sun's Wildfire system used multiple address-interleaved snoop buses supported by a point-to-point data transfer network [38]. Since the data bus does not require broadcasting, the bandwidth of the data bus can be improved easily with point-to-point networks. Therefore, the address bus bandwidth and snoop tag bandwidth limits coherence bandwidth.

In this dissertation, we propose *subspace snooping* to increase the effective snooping bandwidth for future snooping coherence protocols. Subspace snooping increases the snoop tag bandwidth by eliminating the need for snooping all the caches. For each request, subspace snooping allows only a subset of processors to respond to the request, improving the one-to-all snooping to one-to-many snooping. However, subspace snooping still requires scalable bus bandwidth to broadcast requests. For the correctness of subspace snooping, the processors which have valid copies of a block must be guaranteed to snoop on bus requests to the block address.

The advances of optical interconnections for multiprocessors will increase the broadcasting network bandwidth tremendously, providing enough bandwidth

for hundreds of processors. However, traditional snooping protocols require all processors to respond to each request. The snoop tag bandwidth should also scale with the address bus bandwidth. However, the tag bandwidth may not scale due to limited bandwidth and excessive energy consumption of tag lookups. Subspace snooping solves the limited tag bandwidth problem by snooping a subset of processors for each bus request. Subspace snooping mitigates the snoop tag bottleneck when abundant bus bandwidth is available either with either heavily interleaved electrical buses or optical buses.

1.3 Contributions

The previous section presented three solutions to improve distinct aspects of MP memory system overheads. Through the solutions, this dissertation makes the following contributions:

- This dissertation investigates how the limited off-chip bandwidth affects the optimal processing core size and cache capacity in future CMPs.
- This dissertation proposes a bank-based shared cache design for CMPs, supporting configurable sharing degrees. The proposed organization provides shared flexibility as well as improved average hit latency and bandwidth.
- This dissertation evaluates the trade-offs of sharing degrees in CMP caches. The potential benefits of per-application or per-cacheline sharing degrees are also explored.
- This dissertation proposes a CMP cache design supporting dynamic block migration to reduce hit latencies.

- This dissertation proposes a technique which adopts speculation to hide communication latencies. The technique shows that decoupling performance protocols from correctness protocols can improve the performance of systems and simplify the design.

- To use speculation, it is essential to have an accurate prediction mechanism for communication data. This dissertation proposes the first method, to our knowledge, to use stale (invalid) values in the local caches.

- This dissertation identifies that the snoop tag bandwidth limits the scalability of snooping coherence, when the bus bandwidth can be increased with optical interconnects. To remove the bottleneck, the proposed technique decouples the snoop tag bandwidth from the bus bandwidth, allowing only a subset of processors to be snooped for each coherence request.

- This dissertation proposes a dynamic mapping algorithm to adapt the partitioning of processors to current sharing patterns. The mapping algorithm reduces the number processors to be snooped.

To evaluate multiprocessors with commercial applications, we developed a full-system simulation environment with an execution driven timing model. Full system simulation is essential to support commercial applications which have become the most widely-used workloads for multiprocessors.

1.4 Dissertation Organization

In this section, we provide an overview of this dissertation work. Chapter 2 describes the two simulation infrastructures used for this dissertation. First, the *MP-sauce*

simulator is a full-system multiprocessor simulator with out-of-order processor models. It supports commercial workloads with an operating system running on the simulator. This chapter presents the requirements of simulation infrastructures for this dissertation, and describes the architecture and workflows of the MP-sauce simulation environment. The second simulator, *Augmint*, is a trace-based simulator with binary instrumentation. The Augmint simulator, without time-consuming detailed models, allows fast simulation with many processors. This simulator is appropriate to evaluate large scale multiprocessors.

Chapter 3 explores the design space of future CMP L2 caches, focusing on the trade-offs of processor cores and limited off-chip bandwidth. This chapter finds the best configuration of processors and caches under varied area and bandwidth limitations. We show the importance of miss reduction in future CMP cache designs, emphasizing the need for cache sharing.

Chapter 4 presents a configurable design for CMP shared caches. Although cache sharing provides effective miss reduction, it may suffer from long hit latencies due to wire delay. To reduce the hit latencies, We propose a shared L2 cache design with dynamic migration. The migration policies and search mechanisms are discussed. As an alternative solution to dynamic migration, this chapter also evaluates level-one prefetching mechanisms.

In Chapter 5, we propose *coherence decoupling*, which allows speculative execution on communication misses. Speculation can hide long communication latencies in multiprocessors. We show the effect of communication misses with increasing cache sizes. Based upon the observation, we describe the components of coherence decoupling: value prediction mechanism, correctness protocol, and recov-

ery mechanism. The result section presents the accuracy of value prediction and the performance improvement by coherence decoupling.

In Chapter 6, we propose *subspace snooping*, which increases the effective communication bandwidth in multiprocessors. This chapter raises the issue of snoop tag bottlenecks for scaling multiprocessors and current advances in optical interconnects. The snoop tag lookup overheads can limit the scalability of snooping protocols even when a very large interconnection bandwidth is provided. The result section evaluates subspace snooping with the Augmint simulator. Chapter 7 presents the conclusions of this dissertation.

Chapter 2

Simulation Methodology

In this chapter, we present two simulation tools for this dissertation. First, the *MP-sauce* simulation environment provides a full system simulation with detailed timing models for out-of-order processors and caches. We developed the MP-sauce simulator based on SimOS-PowerPC (SimOS-PPC) [95], SimpleScalar [13] and SimpleMP [87]. We use MP-sauce to study coherence decoupling, which requires speculation at processors, and a shared CMP cache design (Coherent Shared NUCA), which requires detailed performance models for processors and on-chip caches. A drawback of this simulation environment is its simulation time and scalability. Due to the detailed timing models, simulation time is long and thus it can not scale to simulate several tens of processors. Furthermore, the number of supported processors is limited to 24 by the AIX 4.3.1 operating system running on the full system simulator.

The second simulator for fast and large scale parallel simulation is the Augmint toolset [82]. The Augmint simulator was derived from the Mint simulator by

adding the support for the x86 instruction set. Unlike the MP-sauce simulator, the Augmint simulator embeds tracing instructions to applications, and generates memory access traces. Timing memory models use the traces to simulate memory system performance. Although the timing accuracy of simulation is inferior to that of MP-sauce since it is not execution-driven, it can provide the fast simulation and scalability the subspace snooping study requires.

In this chapter, we describe the architecture and characteristics of the MP-sauce simulator and the Augmint simulator, and present the workflows to set up and run benchmark applications with the two simulators.

2.1 MP-sauce Simulator

Studying future multiprocessor systems with commercial application poses challenges. To simulate MP systems with the complicated commercial applications, the entire system, including the operating system and I/O devices, must be simulated. Although numerous architectural simulators have been developed and used in the computer architecture research community, only a few recent simulators have supported such full system simulations.

We have developed a full system simulation environment, called *MP-sauce*, based on the SimOS-PowerPC (SimOS-PPC) and SimpleScalar models. SimOS-PPC provides system level services such as device simulators (disk, network and console devices) and facilities for fast simulation (creating checkpoints and resuming simulation from checkpoints). MP-sauce uses the system level service routines for disks, networks, and consoles from SimOS-PPC. Several past approaches for performance simulation with a full-system simulator, use a decoupled approach: A

full system simulator (in our case, SimOS-PPC) generates execution traces as a front-end simulator, and back-end timing models simulate system performance with the traces. Such decoupled simulators may not reflect true speculative execution (wrong-path execution) and non-determistic events in MP systems. MP-sauce uses a more accurate approach than the decoupled simulations. The MP-sauce processor timing models truly execute instructions speculatively and cache coherence models simulate possible transitional states. The integrated approach of MP-sauce is essential to evaluate the effect of speculative execution, which is critical for studying coherence decoupling and future CMPs with out-of-order processors. The development of the simulation infrastructure took two years, including the time spent for adding the benchmark suite to the simulator environment.

2.1.1 Requirements for Timing Simulation for Multiprocessors

There are several requirements for simulation environments to study multiprocessor systems for commercial applications:

- Server workloads use operating system services heavily. Kernel activity is an essential part of studying commercial applications. Furthermore, nearly all commercial applications can run only when full operating system services, such as thread support and I/O support, are available.

- Since modeling the non-deterministic nature of multiprocessors is important to understand inter-processor communication, a true execution-driven simulation is preferred to trace-driven simulations.

- Speculative execution of processing cores needs to be modeled, as target pro-

cessors with out-of-order execution cores exploit more instruction-level parallelism. As discussed before, this accurate modeling of speculation is essential for our studies.

- Realistic coherence protocol simulation with a contention model is necessary. The MP-sauce coherence protocols simulate possible transitional states.

2.1.2 MP-sauce Simulation Architecture

The MP-sauce simulation environment consists of three components: processor and cache simulators, device simulators, and disk image files. Figure 2.1 shows the overall structure of the simulation environment. The AIX 4.3.1 operating system boots and runs on the simulator. The MP-sauce simulation environment consists of three components:

- Device simulation (SimOS-PPC): The MP-sauce simulator uses system-level device simulators from SimOS-PPC. The SimOS-PPC simulator provides simple models for disks, network devices, and consoles [95]. Such hardware devices communicate with the rest of the system through the direct memory access (DMA) routines and interrupt mechanisms. The SimOS-PPC simulator maintains interrupt vectors to forward IO interrupts to appropriate handler addresses. The SimOS-PPC simulator also provides address translation routines, translating virtual addresses to physical addresses.

- Disk image and checkpoints: A disk image is a snapshot of the file systems taken from real systems. They contain all of the necessary files for running the AIX 4.3.1 operating system and applications. The complete hard disk from a

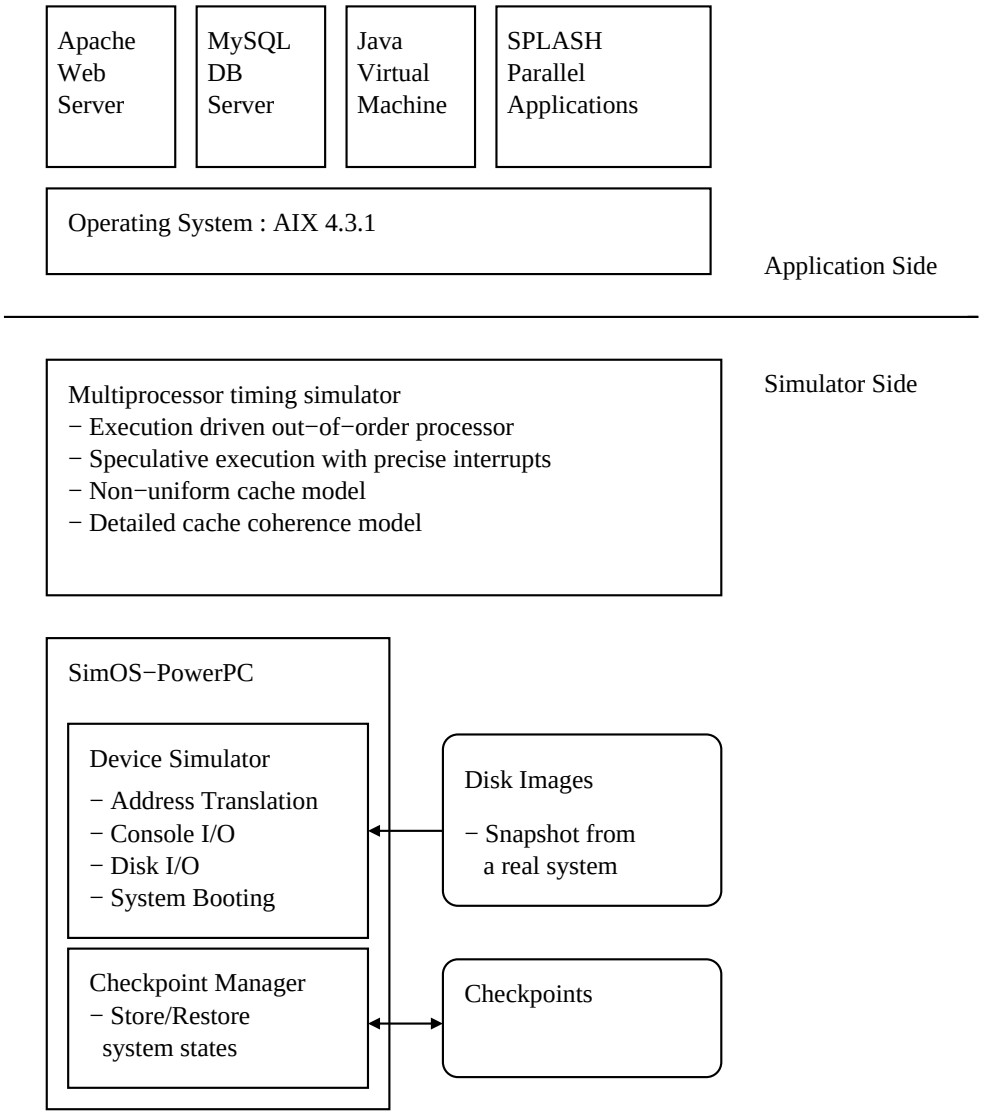


Figure 2.1: MP-sauce simulation environment overview

system with the same operating system is copied into the disk image. In the initialization phase of the simulation, the disk device is mapped to the disk image, and SimOS-PPC can boot operating systems from this disk image.

SimOS-PPC supports checkpoints to reduce the simulation times. For accurate timing simulations, the simulator should measure performance after applications are initialized. Server applications stabilize to optimal operating states after executing a number of requests from clients. Since it takes a long time for warming up servers, checkpoints are created after the initialization and warmup period, and timing simulations are repeated from the checkpointed states. Checkpoint files contain four system states: processor states (register values), memory states, device states (device registers and pending interrupts), and disk updates. Disk updates are changes in disks from the initial disk image. Any disk change does not update the initial disk image. Instead, the changes are recorded in separate files.

- Processor and cache models: The MP-sauce processing core and cache simulators are derived from the SimpleScalar simulator [13] and the SimpleMP simulator [87]. The processor model is a speculative out-of-order execution processor based on the Register Update Unit (RUU), which is a combined instruction issue window, physical register file and reorder buffer [97]. Load and store instructions are tracked in a combined load-store queue.

  The processing core simulates all user-level and kernel-level instructions. The core model implements PowerPC synchronization primitives for multiprocessing. The implementation supports both 32 and 64 bit execution modes for the AIX operating system.

The processor models run truly execution-driven simulation. Register values updated by speculative instructions are buffered in the RUU and update the architectural registers only when instructions are committed. Although loads are issued speculatively, stores are sent to the L1 cache only at the commit stage.

For interrupts, processor pipelines are flushed and start from the new PC obtained from the interrupt vectors. Exceptions are checked when instructions are committed.

The memory simulator models snoop-based multiprocessor caches. Each cache is modeled for non-blocking accesses with MSHR buffers. For accurate modeling, all the transitive states in the MOESI coherence protocol are modeled. For CMP studies, shared caches are modeled with bank-based cache organizations.

2.1.3 MP-sauce Simulation Workflow

Typical simulations with MP-sauce take three steps. Figure 2.2 describes the procedures to set up applications and run simulations.

- *Step 1*: Applications are compiled for target systems and tested on a real system. Commercial applications often include many binaries and complicated directory structures to store configurations and data. Target applications are compiled and installed in a real system as if they are installed for real runs. For server applications, data files are also set up during the installation. For database applications, databases are initialized and actual data are stored.

  After the setup phase, applications are tested on a real system to check the correctness and estimate the running times.

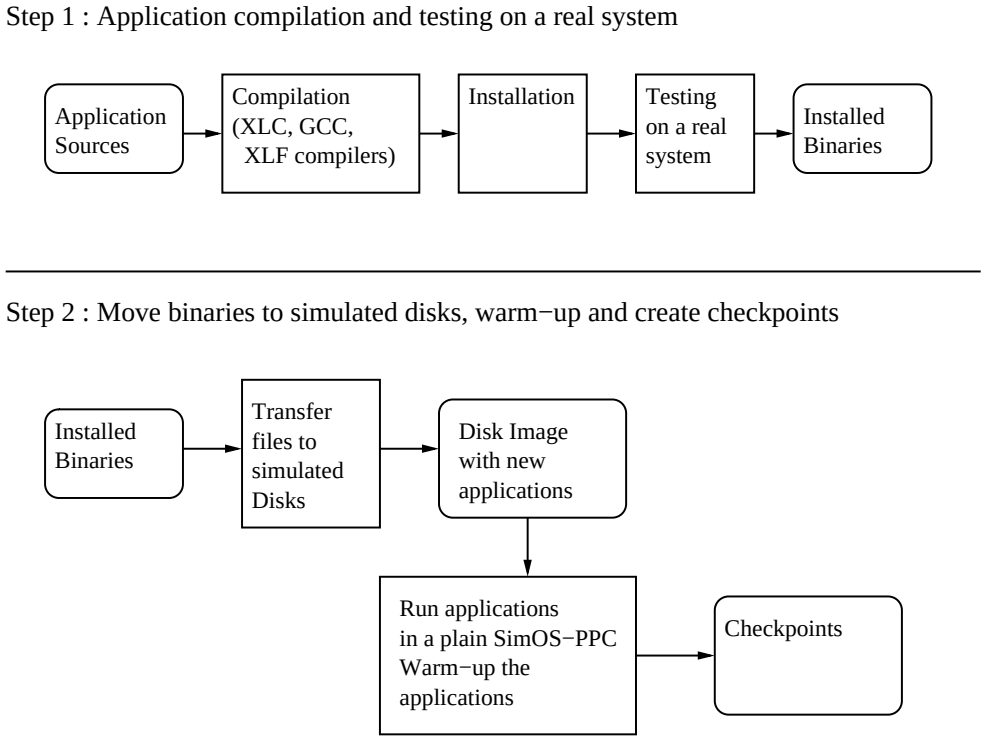


Figure 2.2: Workflow for MP-sauce simulation environment

- *Step 2*: The compiled binaries and necessary files for data and configurations are brought into the simulated system. The SimOS-PPC provides file transfer facilities to move files between the simulated system and the host system.

  After being transferred to the simulated system, applications are run on the SimOS-PPC simulator for warming-up the applications. For this step, we use the SimOS-PPC simulator without the MP-sauce timing models for fast execution. After applications are initialized and warmed up, a checkpoint is created so that timing simulation can start directly from a desired point.

  For server applications, many transactions are executed to warm up server databases. Timing simulation is conducted only after servers are in stable states.

- *Step 3*: Using the checkpoint from Step 2, the MP-sauce simulator runs timing simulation and generates performance analysis outputs. Only this step needs to be repeated to simulate systems with different hardware configurations. However, changing the number of processors in a system requires booting the system, since the OS needs to know the number of available physical processors. Therefore, step 1 and step 2 must be repeated to change the number of processors.

2.1.4 Benchmarks

Our benchmark suite represents medium-sized server workloads and scientific computation workloads. Adding scientific applications to our benchmark suite allows interesting comparisons between server applications and scientific applications. The benchmark suite consists of the following three server-based applications, and shared

memory scientific benchmarks:

- Web server (SPECWeb99): HTTP web servers deliver static and dynamic content to clients. Data file sets and requests are generated according to the SPECWeb99 specification. The requests are mixed for static file accesses and dynamic content requests. The scripts for dynamic content are programmed with the Perl script language. Apache 1.3 is used for HTTP servers with mod_perl 1.27 to process Perl scripts for serving dynamic content.

- E-commerce server (TPC-W): The E-commerce server is derived from the TPC-W specification. The server emulates an on-line book store, where clients can search books by title, author, etc., and can issue orders to buy books. The server consists of three components: a front-end PHP script processor, a web server, and a back-end database server. The front-end servers use Apache 1.3 with the PHP 4.2.3 module. MySQL 4.0 is used for the back-end database system.

- Java business server (SPECjbb2000): The Java business server uses SPECjbb2000. SPECjbb2000 emulates 3-tier server-side Java applications. The server models TPC-C-like warehouse databases with clients on a Java virtual machine. Each client is running as a separate thread, and simultaneously accesses in-memory warehouse databases.

- Scientific applications (SPLASH-2): Our scientific applications are selected from the SPLASH-2 benchmark suite [106]. Thread creation and lock primitives are implemented with the pthread libraries.

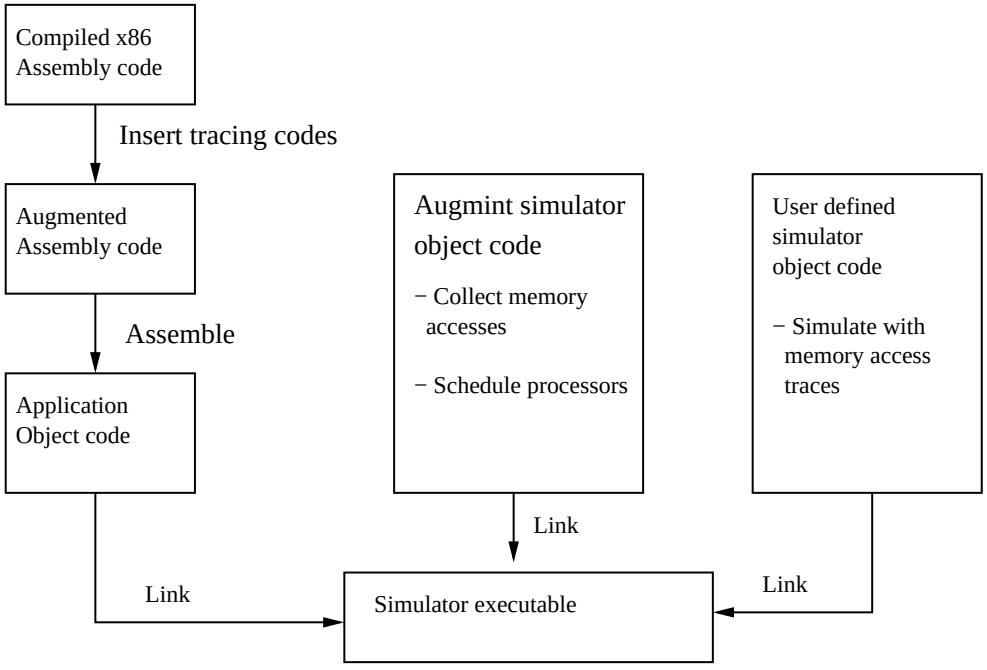


Figure 2.3: Workflow for Augmint simulation environment

2.2 Augmint Simulator for Fast Simulation

The Augmint simulation toolkit runs applications natively on host systems by instruction augmentation [82]. Applications are augmented with tracing instructions, and memory access traces are forwarded to the architectural simulator. The Augmint simulator models simple perfect in-order processors with a cache hierarchy.

The Augmint simulation toolkit sacrifices the accuracy of simulation to reduce the simulation times. The differences between Augmint and MP-sauce are as follows:

- In Augmint, applications are augmented with trace instructions, and linked with Augmint scheduler objects and architectural simulator objects. The final

executable runs natively on host systems.

- Only a perfect in-order processor is supported in the Augment simulator. The processor model can always execute one instruction every cycle if the memory system is perfect. Any delay due to cache misses can be added to the execution model.

- Augmint can run only scientific applications. The applications must use a specific implementation for synchronization (PARMACS macros for thread management, locks, and barriers).

2.2.1 Augmint Simulator and Workflow

Figure 2.3 describes the simulation workflow with the Augmint toolkit. The Augmint toolkits consist of three components:

- Instrumentation tool: adds tracing instructions to the assembly codes of applications. The tracing instructions check memory access instructions and trigger the Augmint scheduler.

- Augmint scheduler: adjusts each processor's cycle advancement. The scheduler forwards memory addresses to the architectural simulator. The architectural simulator calls the Augmint scheduler when the access is finished through the cache hierarchy. Based on the delay from the cache hierarchy, the Augmint scheduler controls the processor scheduling.

- Architectural simulator: The architectural simulator should be provided by simulator users. Using event-based interfaces from the Augmint scheduler, the architectural simulator models the cache hierarchy and coherence mechanisms.

As shown in Figure 2.3, the final application executable is linked with three objects. First, applications are first expanded by macro processing. The applications use PARMACS macros, which define thread management and synchronization. Macro processors expand such macros with the Augmint implementation of thread management routines. Second, the applications are compiled to assembly codes, and augmented with tracing instructions with the instrumentation tool. The instrumented codes are compiled to object files. Third, the Augment scheduler and architectural simulator are linked with the application objects. The final executable can run on x86 systems.

We use scientific applications from SPLASH-2 benchmark suites [106], and *AppBT*. AppBT is a shared memory version of BT in the NAS parallel benchmark suite [14].

Chapter 3

Exploring The Design Space of Future CMPs

Chip-multiprocessors (CMP) have become a promising approach for increasing job throughput in servers, and are moving to desktop computing. With increasing transistor density, it is likely that future CMPs will have considerably larger numbers of processors than today, for two reasons [1]. First, the superscalar paradigm is reaching diminishing returns, particularly as clock scaling has slowed precipitously. Second, global wire delays will limit the area of the chip that is useful for a single conventional processing core. Since a single processing core will be unable to use the bulk of the chip real estate, the additional transistors will likely be used for additional cores.

In this chapter, we present the study of processing cores and cache sizes in CMPs for given area and off-chip bandwidth limitations. We determine the CMP organizations that maximize total chip performance, which is equivalent to

job throughput in this study. We consider the following factors:

- *Processor organization*: Whether powerful out-of-order issue processors, or smaller, more numerous in-order processors provide superior throughput.
- *Cache hierarchy*: The amount of cache memory per processor that results in maximal throughput. The ideal capacity is a function of processor organization, memory latency, and available off-chip bandwidth.
- *Off-chip bandwidth*: Finite bandwidth limits the number of cores that can be placed on a chip, forcing more area to be devoted to on-chip caches to reduce bandwidth demands.
- *Application characteristics*: Applications with different access patterns require different CMP designs to attain the best throughput. Different applications show varying sensitivities to L2 cache capacity, resulting in widely varying bandwidth demands.

These constraints have complex interactions. More powerful processors place a heavier individual load on the off-chip memory channels, but smaller, more numerous processors may result in a heavier aggregate bandwidth load. Larger caches reduce the number of off-chip accesses, permitting more processors to share a fixed bandwidth, but the larger caches consume significant area, resulting in room for fewer processing cores. In this chapter, we study the relative costs in area versus the associated performance gains, showing the organizations that maximize performance per unit area for future technology generations. Since my focus is on performance bounds, we do not consider power limitations in this dissertation.

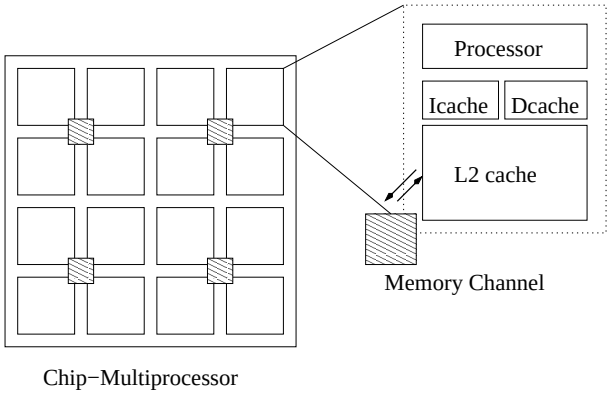


Figure 3.1: Chip-multiprocessor model for chapter 3: private L1 and L2 caches

3.1 Technology models for evaluating CMP alternatives

In this chapter, we focus on throughput-oriented workloads with no sharing of data among tasks to evaluate the area efficiency of chip-multiprocessors. As shown in Figure 3.1, our CMP model has two levels of cache hierarchy, with L1 and L2 caches coupled to individual processing cores for scalability. Deferring the investigation of shared L2 cache designs to the next chapter, we assume private L2 caches for the CMP cache model in this chapter. Each L2 cache is connected to the off-chip DRAM through a set of distributed memory channels. Since the number of memory channels is limited by physical and economic constraints, the allocation of the finite bandwidth must be considered when designing cost-effective CMPs. Thus our models account for time-multiplexing of the memory channels, and we investigate effects of channel contention on the ideal balance between cache and processor area allocations.

3.1.1 Area models

An analysis of area efficiency requires accurate models of processing cores and caches of varied capacities. Gupta et al. have derived a set of technology-independent area models empirically, by measuring die photographs of commercial microprocessors and normalizing the results for feature size [36]. To enable simple area trade-offs among processor core areas and cache bank areas, the model expresses all area in terms of cache byte equivalent area (CBE), which is the unit area for one byte of cache, similar to the Equivalent Cache Transistor metric of Farrens *et al.* [28]. We use a metric expressed in bytes for greater ease in reasoning about processor and cache area trade-offs. The CBE includes the amortized overheads for tags, decoders, and wires, in addition to the 8 SRAM cells.

For our processor model, we considered in-order and out-of-order issue processors ranging from 2-wide to 8-wide issue widths. In Table 3.1, we show the harmonic means of IPC (Instructions Per Cycle) of our benchmarks listed in Section 3, for each model with varying L2 cache size. The number of ALUs are scaled with the issue width. For in-order cores, issue width has little effect on performance, but out-of-order cores have significant performance improvement from 2- to 4-way issue cores. When the performance per unit area is considered, we found 2-way in-order and 4-way out-of-order processors are the most area-efficient models, and chose them as our processing core models. Table 3.2 shows the complete configuration of the two processor models used in this chapter. P_{IN} is a simple 2-way in-order issue processor that is roughly comparable to the Alpha 21064 [72]. The P_{OUT} processor is a more aggressive, 4-way issue out-of-order processor comparable to the Alpha 21264 [51]. These simulated processors have different microarchitec-

	L2 cache size	2-way	4-way	8-way
In-order	128KB	0.20	0.21	0.21
	256KB	0.23	0.24	0.25
	512KB	0.24	0.25	0.25
	1MB	0.27	0.28	0.29
Out-of-order	128KB	0.26	0.31	0.33
	256KB	0.31	0.38	0.40
	512KB	0.32	0.39	0.41
	1MB	0.38	0.47	0.50

Table 3.1: Harmonic means of IPCs for six processor models

	P_{IN}	P_{OUT}
Instruction issue	in-order	out-of-order
Issue width	dual-issue	quad-issue
Instruction window (entries)	16	64
Load/store queue (entries)	16	64
Branch predictor	bimodal (2K)	2 level (16K)
Number of integer ALUs	2	4
Number of floating-point ALUs	1	2
Estimated core area (CBE)	50 KB	250 KB
L1 Instruction cache	32 KB	32 KB
L1 Data cache	32 KB	32 KB
Total area (core + I/D caches)	114 KB	314 KB

Table 3.2: Parameters for two processor models

tures than the Alpha 21064 and Alpha 21264, but are intended to model processors of similar capabilities implemented with similar transistor budgets. The results of this model show that the core area of P_{OUT} is five times larger than that of P_{IN} and with L1 instruction and data caches, P_{OUT} is three times larger P_{IN}.

This chapter assumes a large but fixed sized die of $300mm^2$. With smaller feature sizes, the available area for cache banks and processing cores increases. The primary goal of this chapter is to determine the best balance between per-processor cache area, area consumed by different processor organizations, and the number of

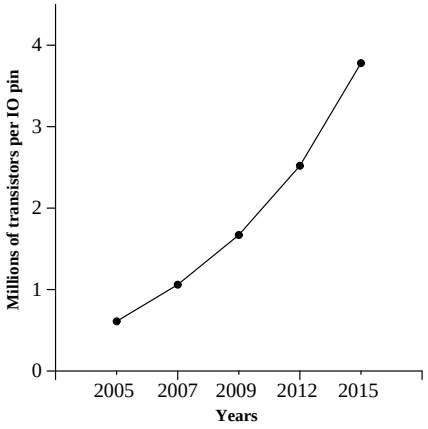


Figure 3.2: Transistor counts per IO pin

cores on a single die.

3.1.2 I/O pin bandwidth

While increasing transistor budgets can accommodate large numbers of processing cores on a single chip, the communication between the chip and the rest of system is both critical for performance and expensive to scale. The number of signal I/O pins built on a single chip is limited by physical technology and does not scale with the number of transistors. Figure 3.2 shows the projected ratio between chip transistor capacity and signal pin count, according to the ITRS 2005 projections [93]. While pin count is increasing, the number of transistors is increasing at a much higher rate. For example, in 2015 there will be 6 times more transistors per pin than in 2005.

Another factor limiting off-chip communication is that, to date, I/O signaling speeds have not increased at the same rate as processor clock rates. It is common today to find a 1 GHz processor connected to memory through a 133MHz back-side

bus. Even though active research aims to improve pin bandwidth by substantially increasing the pin transfer rates into the Gigabit per second regime [23, 27, 111], the disparity between the computation capacity and off-chip bandwidth will persist for the foreseeable future. For our experiments, we scale the chip pin density according to the ITRS projections for signal pin density at a fixed $300mm^2$ die size. We scale the pin speeds linearly with technology at one-half the speed of the processor clock.

3.1.3 Maximizing throughput

In a CMP, the performance on server workloads can be defined as the aggregate performance of all the cores on the chip. For these workloads, two parameters—the number of cores (N_c), and the performance of each core (P_i)—are necessary to estimate peak performance P_{cmp} of a server CMP:

$$P_{cmp} = \sum_{i=1}^{N_c} P_i$$

The performance of an individual core in a CMP (P_i) is dependent on application characteristics such as available instruction level parallelism (ILP), cache behavior, and communication overhead among threads. For applications that spend significant portions of their execution time in communication and synchronization, parallel efficiency of the applications drops precipitously, and realized P_{cmp} will drop below peak P_{cmp}. However, in many server applications, threads are initiated by independent clients, and they rely on relatively coarse-grained data sharing (or no sharing at all), thus resulting in high parallel efficiency.

To simplify our initial study on CMP designs, we focus on the ILP and cache behavior of serial applications, deferring a study of application communication and synchronization effects to the next chapter. Our base assumption in this

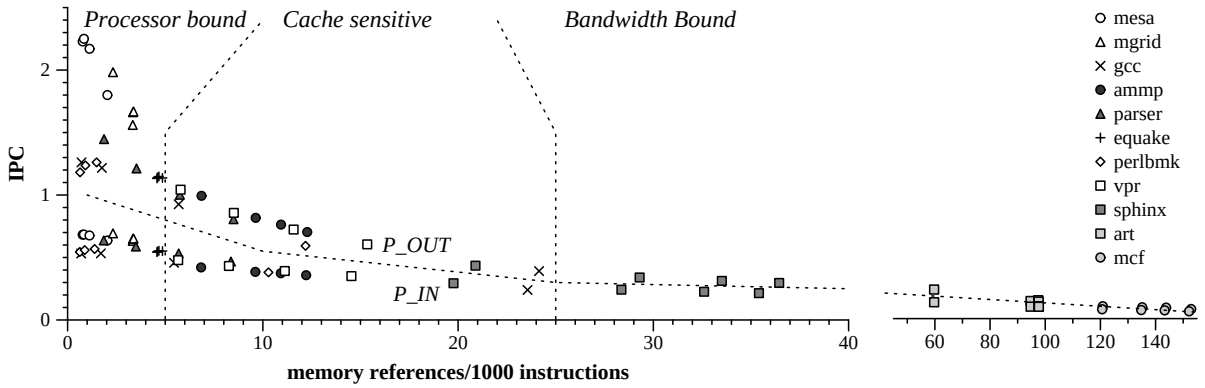


Figure 3.3: IPC versus rate of DRAM accesses.

study is that all processes are independent of one another, which is the case in a multiprogrammed environment. The metric of performance in this chapter is total throughput, measured in instructions per clock (IPC). Given a fixed die size, this metric is equivalent to an area efficiency metric. The optimization goal is to balance the number of cores with the performance and bandwidth demands of individual cores.

3.2 Application characteristics

The best allocation of processor area, cache area, and bandwidth depends on the the characteristics of the applications in the workload. This section characterizes the applications in this study based on their resource demands. We chose ten applications from the SPEC2000 benchmark suite and the *sphinx* speech recognition application [59] to provide a wide range of memory system behavior. The SPEC2000 applications include *mesa*, *mgrid*, *equake*, *gcc*, *ammp*, *vpr*, *parser*, *perlbmk*, *art* and *mcf*. The experimental results show that the applications can be categorized by the following criteria:

- Processor-bound: applications whose working sets are captured easily in the L2 cache, who require few external DRAM accesses, and as a result are largely insensitive to cache capacity and bandwidth restrictions. *Mesa*, *mgrid*, and *equake* are in this class.

- Cache-sensitive: applications whose performance is limited by L2 cache capacity, as larger caches capture increasing fractions of the working sets. *Gcc*, *ammp*, *vpr*, *parser*, and *perlbmk* are in this class.

- Bandwidth-bound: applications whose performance is limited strictly by the rate that data can be moved between the processor and the DRAM. The working sets of the applications are much larger than L2 cache size, or there is little locality in the access patterns. *Art*, *mcf*, and *sphinx* are in this class.

Applications are not bound to one class or another; they move among these three domains as the processor, cache, and bandwidth capacities are modulated. To help characterize the memory behavior of the applications, we use the metric of *DRAM references per thousand instructions*. A DRAM reference results directly from an L2 cache miss or writeback. Consequently, this metric follows directly from the characteristics of the application, the L2 cache capacity, and as shown in Section 3.2.4, the number of processors in the CMP, due to sharing of memory channels by multiple processing cores.

3.2.1 Experimental methodology

We measure instruction throughput and memory behavior using the SimpleScalar tool set [13]. We configured SimpleScalar to model both the in-order and out-

of-order processors, P_{IN} and P_{OUT}, described in Table 3.2. We further modified SimpleScalar for the chip-multiprocessor experiments to run multiple copies of the same application with varying numbers of memory channels and sharing of the channels among the processors. The memory system simulates non-blocking, write-back caches, and bus contention at all levels. The L1 instruction and data caches are two-way set associative with 64-byte blocks, and the L2 caches are four-way set associative with 128-byte blocks. To focus more directly on the larger L2 caches, L1 instruction and data caches are fixed at 32KB. For our benchmarks, the applications show little performance improvement with larger L1 caches, due to projected increase in access delays at smaller technologies. We therefore used the smallest of these equivalently performing cache organizations, since it was the most area efficient.

To simulate the effects of cache size on cache access latency, we used the Cacti tool to determine access latency as a function of cache capacity [92, 105]. Given the cache capacity, associativity, number of ports, and number of data and address bits, Cacti finds the best cache configuration (minimal access time) by modeling a large number of alternative cache organizations. The cache hit latencies of 128KB, 256KB, 512KB, and 1MB are 4, 5, 7, and 9 cycles, respectively.

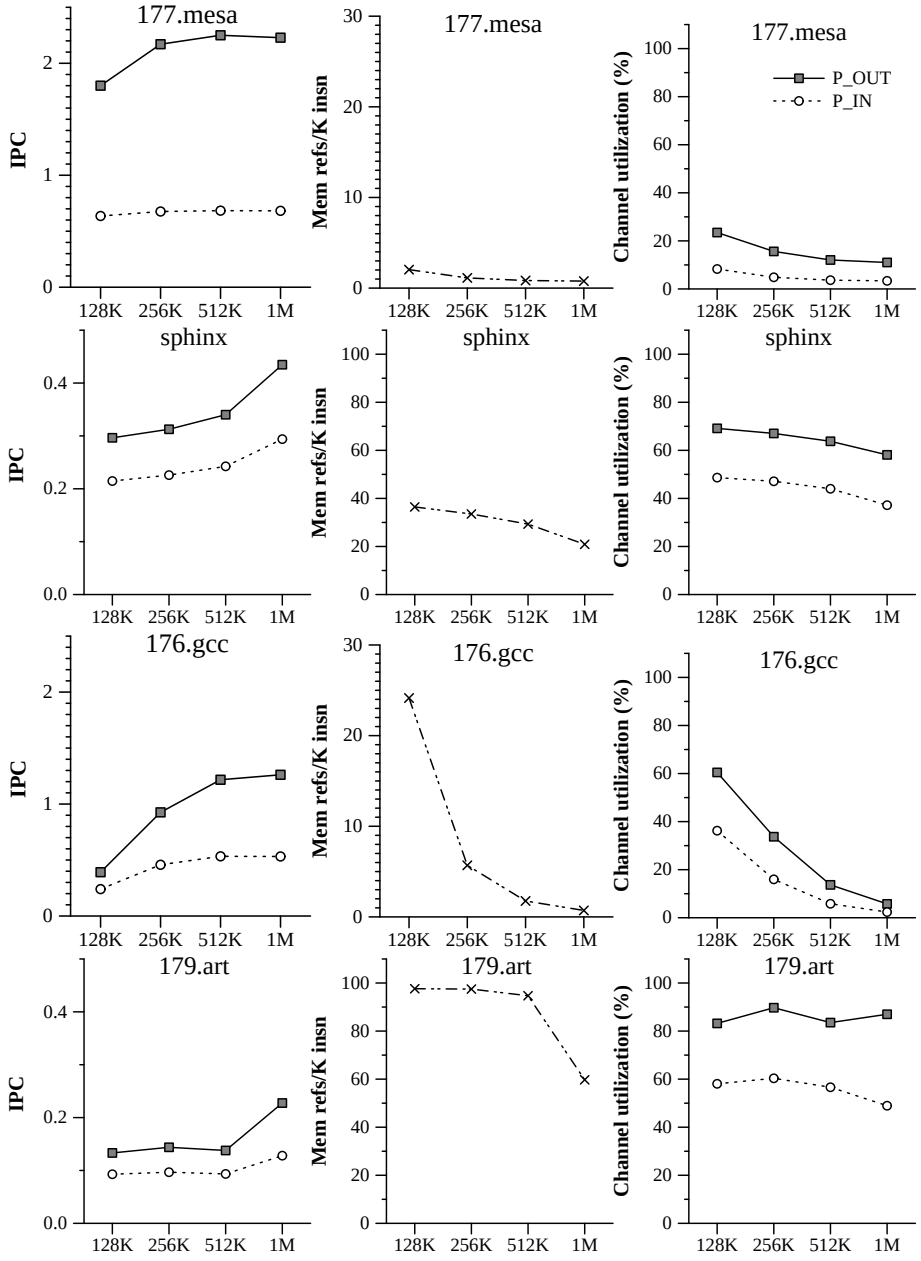
To account for aggressive, next-generation memory system technology, the DRAM portion of our simulator models Direct Rambus memory channels in detail [19]. The data bus is clocked at 400 MHz, and data are transferred on both edges of the clock. A Rambus channel uses 30 pins for control and data signals, with a data width of 2 bytes. If more bandwidth is needed and pins are available, multiple Rambus channels may be used in concert to form a single, logically wider

memory channel. We use two Rambus channels for our memory channel, resulting in a total of 60 pins per channel with a data width of 4 bytes. As mentioned in Section 3.1, the Rambus DRAM clock rate is set to effectively one-half (one-quarter clock with dual-edge transition) that of the processor, and assumes that memory channel speeds will scale with processor clocks for future technologies.

For each application, the first five billion instructions of execution are skipped to avoid simulating benchmark initialization, and the subsequent 200 million instructions are simulated in detail.

3.2.2 Application resource demands

To investigate the uniprocessor memory requirements of the applications, we varied the processor model and L2 cache capacity to modulate the DRAM reference frequency. The resulting instruction throughput is shown in Figure 3.3, as a function of reference frequency. For each application, a family of points is plotted corresponding to the two processor models (P_{IN} and P_{OUT}) and L2 cache capacities ranging from 128KB to 1MB. The general behavior for all of the applications is an increase in DRAM reference frequency as cache capacity is decreased, resulting in a reduction in IPC. Unsurprisingly, the IPC for the out-of-order processor uniformly exceeds that of the in-order processor, although the two organizations converge as the applications become bandwidth bound, with DRAM reference frequencies greater than 25 per 1000 instructions. We note that the benchmarks exhibiting more than 4 references per 1000 instructions show remarkably similar performance as a function of DRAM access rate. The applications can be divided into the three categories based on their position along the x-axis:



(a) Performance (b) DRAM refs (c) Channel util.

Figure 3.4: Effect of varying L2 cache size.

- Processor-bound benchmarks: The applications *mesa* and *mgrid* have few DRAM references per instruction. The IPC for these programs is high, particularly for P_{OUT}. The IPC, as well as the DRAM reference frequency, is largely insensitive to cache capacity. *Equake* exhibits similar behavior, even though its DRAM reference frequency is much larger than *mesa* and *mgrid*. Thus processor-bound applications show relatively high IPC and have working sets small enough to fit into moderately sized L2 caches.

- Cache-sensitive benchmarks: The DRAM reference frequency and performance of *gcc*, *ammp*, *parser*, *perlbmk*, and *vpr* are much more dependent upon the L2 cache capacity. As cache sizes increase, the memory references tend to drop, making these applications appear to be processor-bound, particularly when the cache becomes large enough to hold the current working set. With smaller cache capacities, reference frequency increases and IPC drops substantially.

- Bandwidth-bound benchmarks: *sphinx*, *art* and *mcf* place enormous bandwidth demands on the off-chip interconnect. Even though large L2 caches reduce DRAM reference frequency somewhat, the effective lack of a working set results in low IPC even with the largest 1MB L2 caches. Because the L2 cache hit rate is so low, performance is directly proportional to the available bandwidth.

3.2.3 Processor organization and cache size

As shown in Figure 3.3, uniprocessor performance depends both on the processor organization and cache capacity. However, the effectiveness of increased cache capacity and out-of-order processors is limited by the bandwidth demands of the ap-

plications. To display these characteristics more clearly, Figure 3.4 shows the IPC, DRAM access frequency, and memory channel utilization as a function of cache capacity. Four applications are shown: *mesa* (processor bound), *gcc* (cache sensitive), and *sphinx* and *art* (bandwidth bound).

From this figure, we note the following points. First, the gap between the P_{IN} and P_{OUT} configurations in columns (a) depends on the memory demands of the benchmark. The gap is the largest for the processor-bound benchmark (*mesa*), indicating that out-of-order cores will be more area efficient for that category. For the other benchmark (*gcc*), the performance of the out-of-order and in-order cores converges, as cache size drops and more frequent requests are made to memory.

Second, the data in columns (b) indicate that larger caches cause sharp reductions in L2 misses for the cache-sensitive benchmarks (and for *art* when the cache grows sufficiently large).

Finally, in columns (c) the data show that the out-of-order cores place heavier demand on the channel utilization. That demand results from the P_{OUT} cores moving the same quantity of data across the wires in a shorter time. We also note that the Rambus channels saturate at approximately 80% utilization, due to finite bandwidth on the command buses.

Several working sets are clearly visible in these data. When the L2 cache is increased from 256KB to 512KB, *gcc* shifts from the cache-sensitive category to being processor bound. The miss rate for *art* drops significantly when the L2 cache is increased to 1MB. However, even with that drop, *art* is still bandwidth bound, with over 50 DRAM accesses per 1000 instructions.

We define processor bound as having fewer than 5 off-chip accesses per 1000

instructions, and bandwidth bound as greater than 25 references per 1000 instructions. With that definition, it is clear that only two of the benchmarks shown here could tolerate any significant channel sharing: *gcc* for caches greater than 256KB, and *mesa* for any of the cache sizes that we measured.

3.2.4 Channel sharing

Channel sharing arises when multiple processes are executing simultaneously on different processors. Figure 3.5 plots the aggregate IPC seen by a number of processors sharing a single channel. The data show that the processor-bound job *mesa* exhibits good scaling of throughput with increased numbers of channel sharers, except for those experiments with the smallest (128KB) caches. *Gcc* scales or saturates, depending on whether the cache is large enough to hold its distinct, 400KB working set. The bandwidth-bound jobs *sphinx* and *art* show no improvement as more jobs are added, since their bandwidth is the critical resource and already saturated at one job. We note that again, the performance of the in-order and out-of-order cores converge as applications become bandwidth bound. Once too many processors are sharing a channel, adding more processors no longer improves throughput; that area would be better spent increasing the sizes of the caches and reducing the load on the channel. It is exactly that area/performance trade-off that we evaluate in the subsequent section. The utilization of the channel matches the throughput scaling; when the channel starts to reach saturation, throughput levels off.

In Figure 3.6, we plot the utilization of the channel that is shared by one to eight processing cores in *gcc*. When the channel in *gcc* becomes saturated for 128KB cache, utilization drops as more sharers are added. This counterintuitive result

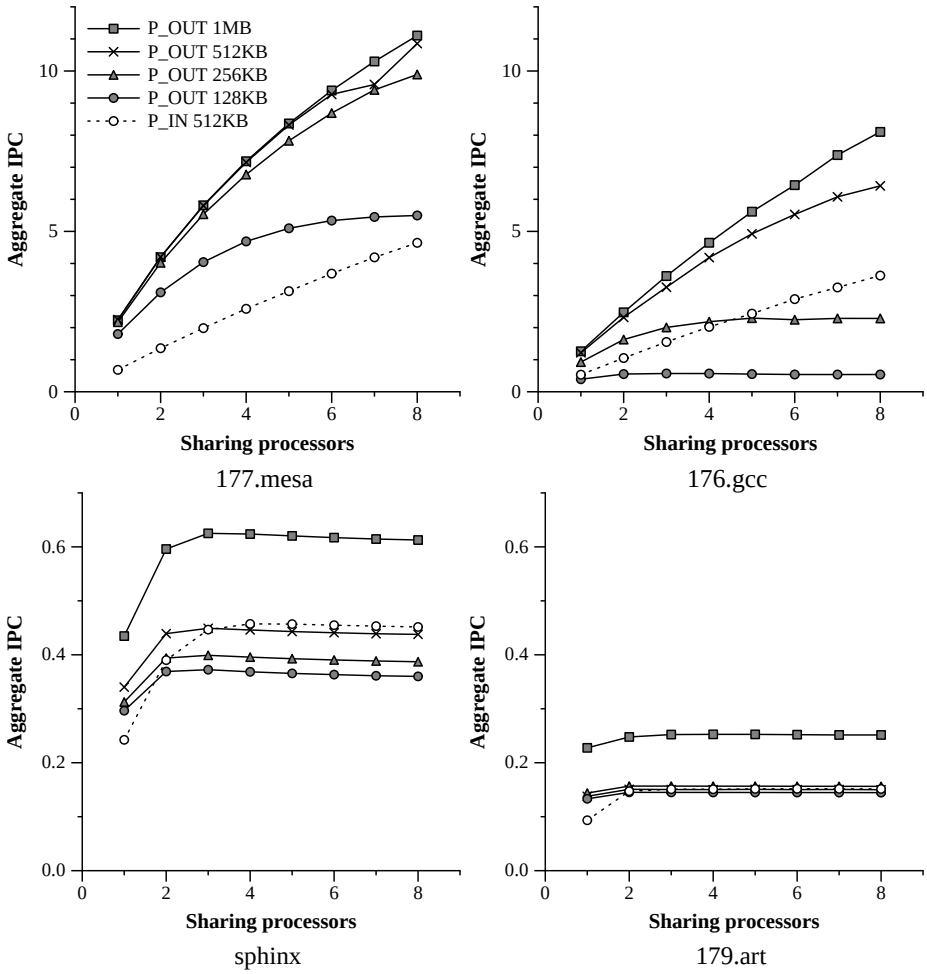


Figure 3.5: Performance scalability versus channel sharing.

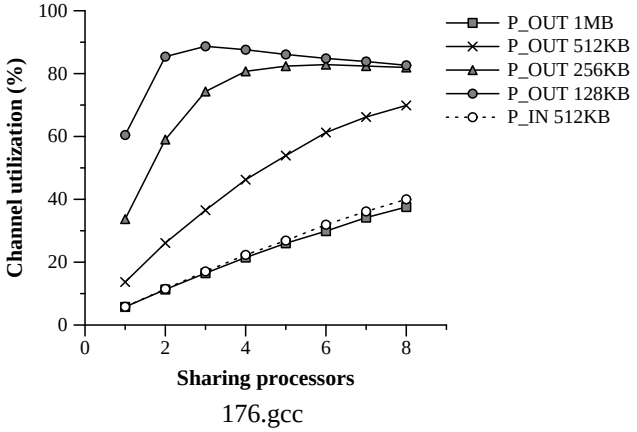


Figure 3.6: Channel utilization (%) versus channel sharing.

occurs because of decreased row buffer locality in the DRDRAM banks. Increased row misses cause gaps in the Rambus command bus schedule, which manifest as slightly lower data channel utilization.

3.3 Maximizing CMP throughput

In the previous section, the results showed that our applications put a widely varying load on the memory subsystem, and that total job throughput levels off when the off-chip bandwidth becomes saturated. In this section, we combine our area analysis with performance simulations and our technology projections to determine which CMP configurations will be the most area-efficient for a future technology generation.

On the left half of Table 3.3, we show the number of processing cores that will fit on a $300mm^2$ chip built in the technology projected in 2010 (a total of 44.0 million CBE). We assume 40% of chip area is devoted to the on-chip interconnections,

memory controllers, IO, clocking etc (26.3 million CBE for cores and caches). As the per-processor caches grow larger, the relative differences between the areas for the P_{IN} and P_{OUT} processors decline. With 128KB L2 caches, 108 P_{IN} cores and 59 P_{OUT} cores can fit on a chip, but with 1MB caches, the number of cores drops to 23 and 19, respectively. On the right half of Table 3.3, we show the number of processor cores that share a channel for each organization. For the 128KB, P_{IN} processors, there are over 7 processors sharing each channel, but for large-cache designs, the number of sharers drops to one.

In Figure 3.7, we show how the number of cores, number of channel sharers, and cache sizes affect area efficiency. The y-axis measures total instructions per cycle across all of the processing cores on the chip, which is equivalent to performance per unit area (since the area is held constant in all experiments). We model non-integer numbers of channel sharers by having some processors share more channels than others. We do not simulate every processor on the chip, but instead simulate just enough to compute the IPC for a subset of the processors, and then scale that result to represent chip-level throughput; our the assumption is that all processors are running the same job, albeit at skewed intervals.

The four graphs in Figure 3.7 show, from left to right, the total chip through-

	No. of cores		Cores/channel	
L2 cache size	P_{IN}	P_{OUT}	P_{IN}	P_{OUT}
No L2	230	83	16.0	5.9
128KB	108	59	7.7	4.2
256KB	71	46	5.0	3.2
512KB	42	31	3.0	2.7
1MB	23	19	1.6	1.4

Table 3.3: Number of cores and cores/channel with varying cache sizes

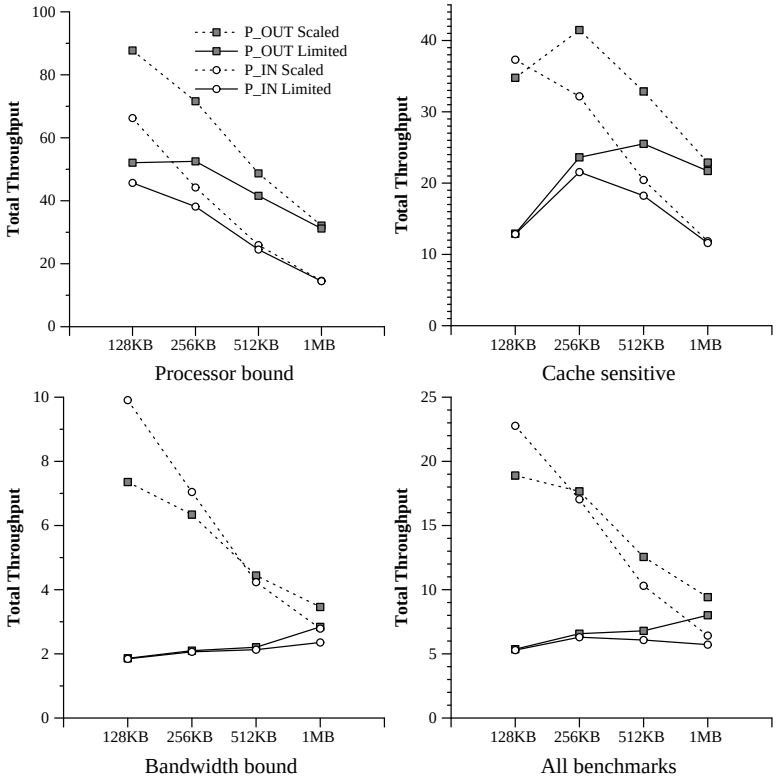


Figure 3.7: Best configurations processor bound, cache sensitive, and bandwidth bound workloads

put in IPC for each of the three benchmark classes (processor bound, cache sensitive, and bandwidth bound) and the total across all benchmarks. The IPCs are computed as the harmonic means of the total chip throughput for the benchmarks in that class, at each design point. We show the means for each benchmark class, since the harmonic mean IPC across all benchmarks is heavily skewed by the low IPCs of bandwidth-bound benchmarks. On each graph, we show separate lines for P_{OUT} and P_{IN}, and also show the effects of two bandwidth capacities. The first, called *limited channels*, fixes the number of pins according to the ITRS projections, and

divides 4-byte wide, 60-pin channels up among the processors on the chip (channel sharing). The second model, called *scaled channels*, assumes that the processor pin counts can be scaled to provide one 60-pin channel per processing core, no matter how many cores exist on the die.

For the processor-bound benchmarks, the most area-efficient configuration uses P_{OUT} cores with 256KB L2 caches. Since those benchmarks are largely compute-bound, the additional cache provides insufficient benefit to justify the area it consumes. With scaled channels, however, the organization that achieves peak throughput is P_{IN} with 128KB caches. That organization, however, would contain 108 processing cores, requiring impossibly many 6480 IO pins.

The cache-sensitive applications show a different result, with the best configuration using P_{OUT} cores with 512KB L2 caches. Enough of the working sets are contained in the L2 caches at that point to make larger caches not worth the additional area consumed.

For the bandwidth-bound applications, the configuration using P_{OUT} processors with 1MB caches is best. However, the difference between P_{OUT} and P_{IN} chip throughput is small and constant across all cache sizes, since the applications are bandwidth-bound at all of the measured cache sizes for both types of cores. We note that the scaled channel throughputs are significantly higher than the finite pin results for the smaller cache sizes, because scaled bandwidth removes the bottleneck from the bandwidth-bound applications. Finally, across all applications, we see that the P_{OUT}, 1MB L2 combination is best for finite bandwidth. However, if each processor could have its own memory channel, regardless of the number of processors, we note that the P_{IN}, 128KB L2 organization would be best.

As shown in this section, the off-chip bandwidth will grow more important as feature sizes decrease and more cores can fit in a chip. In CMPs, reducing off-chip accesses is much more critical for the performance than in uni-processors, since the limited bandwidth is shared by multiple processors.

3.4 Summary

Ideally, highly parallel CMP designs will offer linear scaling of throughput with increasing transistor count. However, limited off-chip bandwidth will always constrain the maximum number of cores that can be placed on a chip. A pressing question for CMP designers concerns the severity of limited bandwidth. In this study of the CMP design space, we have observed the following:

- Transistor counts are projected to increase considerably faster than pins, and there will be 6 times fewer pins per transistor at the technology in 2015 than in 2005. If transistor count increases are used to increase the processor count, the number of pins per processor will decrease. Left unaddressed, that growing imbalance will drastically limit the number of cores that can be used in future technologies, and/or the throughput that can be obtained from those cores.

- Out-of-order issue cores are more area-efficient than in-order issue cores. The area ratio of P_{OUT} to P_{IN}, including 256KB L2 caches, is 1.54. Since P_{OUT} typically provides more than a 54% performance increase over P_{IN}, the out-of-order cores are more area-efficient, unless the application in question is bandwidth bound.

- For the workloads we studied, the impact of insufficient bandwidth causes the

throughput-optimal L2 cache sizes to be 1MB at the technology projected in 2010. The channel contention is sufficiently severe that P_{OUT} cores with 1MB caches are more area-efficient than organizations with significantly smaller caches.

- Applications show remarkably similar behavior and performance when measured against the rate of off-chip accesses. This observation may prove useful for estimating or modeling overall performance of a CMP on heterogeneous workloads, as a function of bandwidth demand.

The methodology of this study has some weaknesses. We are using SPEC2000 benchmarks instead of "typical" server workloads, such as web request processing or database accesses. While those workloads may have large data footprints, the results may not be qualitatively different, in terms of area efficiency, than those of the SPEC2000 benchmarks. Using SPEC2000 benchmarks has given us the flexibility to measure CMP throughput with numerous processor configurations. Since server workloads require the fine-tuning of applications for a fixed processor configuration, we will measure server workloads in more controlled configurations in the next chapter.

In the long term, a tremendous number of processors can be designed on future CMPs to enable scaling of throughput with technology. However, setting the cache hierarchy, and number of cores *a priori* will result in poor performance across many application classes. Future CMPs would benefit from mechanisms to support adaptation to an application's available parallelism and resource needs. In the next chapter, we propose a configurable CMP architecture, which exploits the *application adaptivity*.

In this chapter, we showed how processing core designs, on-chip cache sizes, and off-chip bandwidth affect the CMP throughput. We also evaluated how the application characteristics interact with the three factors. We observed the off-chip bandwidth is one of the most important resources in future CMPs. To reduce the unnecessary waste of the bandwidth, it is crucial to decrease cache misses in CMPs. In the next chapter, we explore the shared cache design space as a miss reduction technique for CMPs.

Chapter 4

Reconfigurable Shared Cache Design for CMPs

In Chapter 3, we showed that reducing cache misses (off-chip accesses) is crucial to improve the throughput of CMPs. Traditional miss reduction techniques such as increasing capacity and associativity, will decrease the CMP cache misses, but in this chapter, we will focus on a new design aspect associated with CMPs: *Cache Sharing*. Unlike uni-processor L2 caches, CMPs allow on-chip processors to share cache capacity. Shared L2 caches for CMPs provide several benefits over private caches:

- *Shared working set*: Shared data are not duplicated in multiple places. Within a shared cache, only one copy of data can reside, thus eliminating unnecessary waste of cache capacity to store duplicate blocks.
- *Prefetching effect*: One processor may fetch cache data from external memory, and the cache blocks can be reused by other processors.

- *Dynamic capacity sharing*: Cache capacity can be utilized more efficiently when demands from processors are not uniform. The cache capacity is dynamically reassigned to multiple processors, adapting to the changing demand.

- *Fast on-chip communication*: On-chip processors communicate with each other through the shared cache. Since the communications do not need to pass through private L2 caches connected with coherence protocols, fast communication is possible with the shared cache.

However, shared caches may suffer from a very long hit latency and high access traffics issued by multiple processors. The latency problem will get worse, as wire delay dominates cache performance and more processors can be integrated in a single chip. Increasing the shared cache sizes to add more processors may result in large but slow caches. This increased latency problem may grow larger than the benefits of shared L2 caches.

In this chapter, we investigate the trade-offs of shared cache designs for future CMPs. Our shared cache design adopts Non-Uniform Cache Architecture (NUCA) to shared caches for CMPs [53]. The shared NUCA provide flexible reconfigurability to evaluate adaptive sharing. We show that hit latency increase is the most significant disadvantage of shared caches. To mitigate the latency increase, we evaluate dynamic block migration from Dynamic NUCA (D-NUCA) and L1 prefetching as an alternative solution to D-NUCA.

We explore this range of caching and sharing policies for a CMP targeted at 45 nanometer technologies, with 16MB of on-chip cache and 16 high-capability processors. The specific layout we propose, based on a static non-uniform cache architecture (NUCA) design, mitigates the effects of growing wire resistivity and

thus intra-cache communication delays, and permits any degree of cache sharing in this single implementation. The underlying hardware is sufficiently flexible to have its degree of sharing chosen dynamically, adjusted by the operating system.

4.1 Related Work

Shared Cache Designs: Shared caches have been studied in the context of chip multiprocessors and multithreaded processors. Sohi and Franklin's early study proposed the interleaved banks for extra ports in private L1 caches, which resembles multi-banking in shared caches [96]. Nayfeh et al. investigated shared caches for primary and secondary caches on a multi-chip module substrate with four CPUs [80]. They examined how the memory sharing patterns of different applications affect the best cache hierarchy. Subsequent work from the same authors showed the trade-offs of shared-cache clustering in multi-chip multiprocessors [81]. With eight CPUs, they observed for private L2 caches, a coherence bus becomes the performance bottleneck, suggesting shared caches to reduce the bus traffic.

Recent studies considered wire latency as a primary design factor in CMP caches. Beckmann and Wood compared three latency reduction techniques including D-NUCA for CMPs with an 8-CPU shared cache [9]. Their study fixed the sharing degree to 8 and observed that combining the three latency reduction techniques can decrease the L2 hit latencies of CMPs. With NuRAPID-based CMP L2 designs, Chishti et al. studied optimizations to reduce unnecessary replication and communication overheads [17]. Speight et al. studied how CMP L2 caches interact with off-chip L3 caches and how on-chip L2 caches temporarily absorb modified replacement blocks from other caches [100].

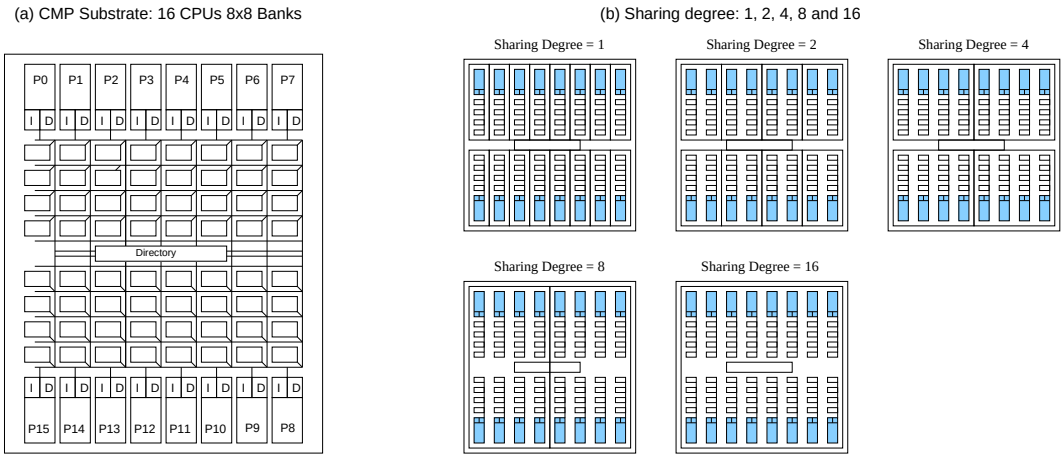


Figure 4.1: 16 processor CMP substrate with reconfigurable sharing degrees

Dynamic Partitioning: Several studies investigated dynamically re-allocating cache capacity for CMPs and multithreaded processors. Suh et al. studied a monitoring system for an optimal dynamic partitioning [102], and a hardware partitioning mechanism for set-associative caches [103]. Liu et al. proposed achieving dynamic bank allocation by re-mapping the banks [64]. Iyer proposed a priority-based cache management systems to allocate cache resources by OS-assigned priority [46]. To prevent thread starvation due to cache capacity sharing, Kim et al. investigated fairness issues in CMP cache sharing [54].

4.2 CMP L2 Cache Design Space

Current CMPs can hold relatively small numbers of processors (two or four per CMP), so one shared cache is shared by all on-chip processors. However, future server-class processors are likely to contain large numbers of processors along with large caches. As the number of processors increases, the shared cache may become

too large and slow if all processors share a single shared cache.

For a CMP, the level-two cache may be shared by all processors, or may be separated into private per-processor partitions, or any point in between. The tension between a greater versus lesser degree of sharing is driven by the reduced misses of greater cache sharing versus the reduced hit latencies of lesser cache sharing. More precisely, we call the *sharing degree* the number of processors that share a given pool of cache. In this terminology, a sharing degree of one means that each processor has its own private L2 partition, whereas a sharing degree of sixteen means that all processors are sharing a single large cache array.

Greater sharing degrees reduce misses in two ways. First, they reduce the number of shared copies of a single line existing on-chip, since each line maps to only one place in shared caches. Second, they provide a larger shared pool to tolerate imbalances across the sharers' working sets. However, larger sharing degrees result in longer cache latencies, as the shared cache is larger than the individual private partitions, assuming that the total cache area is held constant. An ideal design would somehow capture the benefits of both reduced misses *and* reduced hit latencies

In this chapter, we evaluate how sharing degree affects the performance of CMPs, considering wire delay and limited bandwidth in future CMPs. We also discuss how different memory access patterns may change the optimal sharing degrees for different applications.

We propose an on-chip L2 cache architecture for CMPs with reconfigurable sharing degrees. Our CMP L2 cache consists of multiple independently operating banks with configurable switched networks. The banks can be reassigned to different sharing degrees by applications or operating systems. Changing sharing degrees

takes long cycles to flush the on-chip cache. To support different sharing degrees, coherence mechanism must be flexible. We propose a two-level on-chip coherence mechanism based on directory protocols, supporting flexible sharing.

4.2.1 Baseline CMP L2 Cache Organization

As shown in Figure 4.1, the floorplan we evaluate consists of a single large cache array, broken into numerous banks which are connected by a lightweight, switched 2-D mesh network. Processors span the top and bottom edges of the cache array, and each has a port into the L2 cache network. By adjusting the bits used to route memory addresses to a cache bank, the cache array is configurable by the system to use any degree of sharing. If each processor maps the same address bit string to a different bank, the sharing degree is one. If all processors map the same address bits to a single bank, the sharing degree is sixteen.

Figure 4.1 (a) shows a multiprocessor chip projected for 45nm process technology in 2010. The chip has 16 processors and 16MB of L2 cache capacity on a die area of approximately $300mm^2$. The L2 cache resides in the middle of the chip and contains an array of banks connected by a lightweight routing network. Half of the 16 CPUs are at the top and the other half are at the bottom. Because of the large number of CPUs and cache banks, there are many possible design options for the L2 cache. We search for the cache sharing organization that gives the best average performance across a range of commercial and scientific applications.

We first measured the optimal bank size and sharing degree for a cache in which the mapping of lines to banks does not change (this uniprocessor organization was called static NUCA, or S-NUCA, in prior work [53]). We then evaluated per-

Parameter	Value
Processor frequency	10 GHz
Issue width	4
Window size	64-entry RUU
Number of CPUs	16
L1 I/D cache	32KB, 2-way, 64B block, 8 MSHRs
L2 cache	16x16 banks
L2 cache bank	64KB, 16-way, 5 cycle latency
Network	1 cycle latency between two adjacent banks
On-chip directory	10 cycle access latency
Main Memory	260 cycle latency, 360 GB/s bandwidth

Table 4.1: Simulated system configuration

application sharing degrees, as well as different techniques, such as intra-cache line migration (called dynamic, or D-NUCA caches in prior work) and level-one prefetching, to reduce average L2 hit latencies and thus support larger sharing degrees.

With multiple banks per L2 cache, we have the choice of either always putting a block into a designated bank or allowing a block to reside in one of multiple banks (but not simultaneously).

In static mapping, a fixed hash function uses the lower bits of a block address to select the correct bank. L2 access latency is proportional to the distance from the issuing L1 cache to the L2 cache bank. By allowing non-uniform hit latencies, static mapping can reduce hit latencies of traditional monolithic cache designs, which fix the latency to the longest path [53]. Because a block can be placed into only one bank, its L2 access latency is essentially decided by its address. If frequently accessed blocks happen to map to banks with longer latencies, static mapping will not provide optimal performance.

Application	Dataset/Parameters
SPECWeb99	Apache web server, file set: 230MB
TPC-W	185MB databases using Apache & MySQL
SPECjbb	IBM JVM version 1.1.8, 16 warehouses
Ocean	258×258 grid
Barnes	16K particles
LU	512×512, 16×16 blocks
Radix	1M integers

Table 4.2: Application parameters for workloads

4.2.2 Methodology

We evaluated our CMP cache designs using the MP-sauce full-system simulator. Table 4.1 shows a summary of the main architectural parameters. We used three commercial applications: SPECWeb99, TPC-W, and SPECjbb, and four scientific shared-memory benchmarks from the SPLASH-2 suite [106]: Ocean, Barnes, LU, and Radix. Table 4.2 shows the dataset size and other notable features of each application.

As previously discussed, we model an invalidation-based cache coherence protocol in the CMP. The L2 caches are the points of L1 coherence and maintain sharing vectors for L1 caches. The L2 cache bank array is embedded with a 2D-mesh point-to-point interconnection network comprised of links and switches. All messages for coherence and data migration are modeled to consume network bandwidth; however, we assume infinite buffering at each switching node. A centralized directory is used to track cache lines that are cached in the CMP and is consulted on a cache miss to enforce coherence and/or detect misses.

4.3 Performance Effect of Sharing Degree in CMPs

Shared L2 caches have been adopted as an alternative to traditional private L2 caches for CMP L2 cache designs [8, 39, 48, 104]. One of the key factors in CMP shared caches is the number of CPUs that can share a L2 cache, called the *sharing degree* (SD for short). A sharing degree of N means that N CPUs share a L2 cache. In this section, we discuss the trade-offs of higher and lower sharing degrees, and present the performance results with varying sharing degrees.

4.3.1 Trade-offs of higher vs. lower sharing degress

The basic trade-offs of varying the sharing degree are hit latency, hit rate, inter-processor communication, and coherence maintenance overhead. In general, for hit latency, a lower sharing degree is better as each L2 cache is smaller. For hit rates and inter-processor communication, higher sharing degrees are better because they make more efficient use of cache capacity. Since a higher sharing degree has more L1 caches sharing an L2 cache, it is more expensive to maintain L1 coherence. However, a lower sharing degree means more discrete L2 caches on a chip, making maintaining L2 coherence more expensive.

The main advantage of higher sharing degrees is higher L2 cache hit rates. If the working sets across CPUs are not well balanced, private L2 caches can make one CPU suffer from capacity misses while other CPUs have unused cache space. Shared caches, on the other hand, allow that otherwise unused cache space to be used by the space-hungry CPU. Furthermore, shared caches keep at most one copy of a block, not wasting space by storing multiple copies of the same block, unlike private L2 caches sharing copies of the same line. As a result, shared caches can effectively

store more data, indirectly increasing hit rates. A shared cache also avoids the L2 coherence misses generated by private L2 caches [44].

Inter-processor communication through a shared L2 cache is normally faster than that through private L2 caches connected by a coherent bus. With shared L2 caches, processors communicate through L2 cache blocks directly. With private L2 caches, processors have to communicate through private L2 caches and coherence fabrics.

The main drawback of a higher sharing degree is the potential for higher average hit latency due to the larger size, longer wire delays, and increased bandwidth requirements. In future wire-dominated implementations, the effect of increased hit latency may outweigh the benefit of increased hit rates.

Another overhead of shared caches is that each L2 cache needs to maintain coherence for the L1 caches sharing the L2 cache. In this study, the system maintains L1 cache coherence by embedding sharing status vectors in the L2 tag arrays. The tag of an L2 cache line includes a bit mask to indicate which L1 caches have copies of the line. When an L2 cache receives an update request from an L1 cache, it sends invalidation messages to other L1 caches that have a copy of the requested block. Such directory-based L1 coherence was used in the Piranha CMP [8]. Although broadcast-based protocols can reduce the area overhead for sharing vectors [48, 104], we used the directory protocol because it is more scalable for these types of systems, demanding less L2-to-L1 bandwidth than broadcast protocols.

A lower sharing degree means that each chip has more private L2 caches, thus it is more difficult to maintain L2 data coherence. As with the L1 caches, we use a centralized L2 tag directory to maintain on-chip L2 coherence. When an

L2 miss is detected, the request is sent to the centralized L2 tag directory. The directory decides, without snooping other L2 caches in the chip, whether to get data from another L2 cache on the chip or whether to issue an off-chip memory request. The directory-based coherence protocol has a number of advantages over broadcast-based protocols. First, it relieves the coherence bus from becoming the bottleneck. Second, it detects on-chip cache misses faster because it does not need to snoop other L2 caches in the chip. Third, the directory functions as the single external coherence snoop point for requests from other chips, avoiding the need to have multiple L2 caches on the same chip snoop the global bus.

4.3.2 Cache Organization to Support Reconfigurable Sharing Degrees

Shared caches require more bandwidth than private caches, since the data request rate is proportional to the number of processors. Two common ways of increasing bandwidth are i) adding more access ports and ii) splitting a monolithic cache into multiple independently operated banks. L2 caches typically use the latter because it is more cost-effective than having multiple ports. For instance, the dual-processor Power4 CMP [104] uses three banks, and the eight-processor Piranha CMP uses eight banks for each L2 cache.

As exemplified in Figure 4.1, a CMP chip can consist of many independently operated L2 cache banks. To enable high-speed clocking and reduce the space for wires, we use a switched network, instead of traditional dedicated wires, to connect cache banks and processors. A processor may access only one L2 cache directly and must use the coherence fabric to access other L2 caches. Figure 4.1 (b) shows five

Sharing Degree (SD)	SD=1	SD=2	SD=4	SD=8	SD=16
L2 hit time (cycles)	16.5	18.0	20.9	30.0	35.9
Norm. Remote L2 hits	1.00	0.75	0.60	0.36	n/a
Norm. Mem. accesses	1.00	0.88	0.78	0.78	0.66

Table 4.3: Average L2 hit times, and normalized remote L2 hits and memory accesses

possible partitioning schemes in a 16 processor CMP that have sharing degrees of 1, 2, 4, 8, and 16, respectively. With a sharing degree of 1, there are sixteen 1 MB caches, each of which is private to one processor. With a sharing degree of 16, there is only one 16 MB cache, which is shared by all 16 processors.

To optimize bank organization, we evaluated 5 different bank sizes: 64KB, 128KB, 256KB, 512KB, and 1MB. We estimated bank access latencies using Cacti [94] and a wire delay model [2]. Network hop delays are derived from the dimension of banks, with switching overheads. Among the five bank configurations, our experiments show that the 64KB bank size has the best performance across all experimental configurations. For the remainder of this chapter, we assume a 16x16 64KB bank array.

To change sharing degrees, the bank mapping tables in L1 cache controllers, bank controllers, and the central on-chip directory are updated by the operating system. For fully reconfigurable sharing degrees, the sharing status vectors both in L2 tags and the central directory should have enough bits to represent all processors in the CMP. In this substrate, the size of bit vectors in L2 tags and the central directory is the maximum 16.

4.3.3 Results: Finding the Best Sharing Degree

In the baseline CMP with S-NUCA based shared caches, the sharing degree changes the effective L2 hit latency, communication frequency among shared caches (e.g., cache-to-cache transfers), and external memory accesses. Table 4.3 shows the trends in average L2 hit latencies, the number of remote L2 hits and memory accesses as the sharing degree (SD) is changed, with the latter two normalized to the SD=1 case. As the sharing degree increases, shared cache hit latencies increase monotonically from 17 cycles to 36 cycles.

Remote L2 accesses can occur in two cases: First, a read-only shared line is evicted from a local L2 cache, due to a conflict/capacity miss, and upon the next access that line is provided by another shared cache, which has an intact copy of that line. Second, a producer-consumer line is invalidated by another processor in a remote shared cache. In both cases, communication across shared cache boundaries is more expensive in terms of latency and energy consumption than intra-shared cache communication. After detecting a miss in the local shared cache, a miss request is sent to the centralized on-chip directory. The on-chip directory re-transmits the request to another shared cache to service the miss if the cache line is on-chip. Such remote L2 accesses (cache-to-cache transfers) decrease as the sharing degree increases since the likelihood that shared cache lines fall within the same L2 cache increases.

Memory accesses also decrease as the sharing degree increases. With a high sharing degree, processors can share the cache capacity dynamically, reducing the effect of a temporal working set imbalance, resulting in fewer capacity misses than smaller caches. Furthermore, by reducing the number of replicated copies of shared

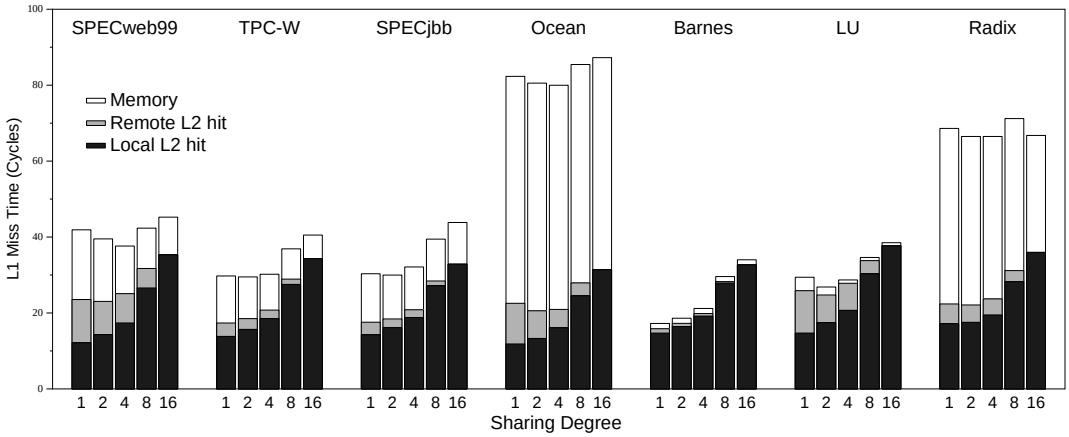


Figure 4.2: L1 miss latencies with varying sharing degrees (16x16 banks)

data, the limited on-chip cache capacity can be more efficiently utilized. As shown in Table 4.3, increasing the sharing degree to 16 can reduce the external memory accesses by 33% on average.

In several ways, the sharing degree affects L1 miss penalties. Figure 4.2 shows a breakdown of the average L1 miss penalty for each application as the L2 cache sharing degree is varied. L1 misses are served by either the local L2 cache, a remote L2 cache, or external memory. Each bar in the graph shows how much latency each component contributes to the average L1 miss penalty.

Across all benchmark applications, the L2 hit time component (black bar) increases monotonically as the sharing degree increases, due to the wire delay increase in the larger caches. The remote L2 and memory components decrease as the degree of L2 sharing is increased; however, those reductions in latency are often not enough to outweigh the increase in local L2 hit latency. For applications that have high local L2 hit rates and low sharing of cache lines like Barnes-Hut, an increased sharing degree beyond private L2 caches degrades performance.

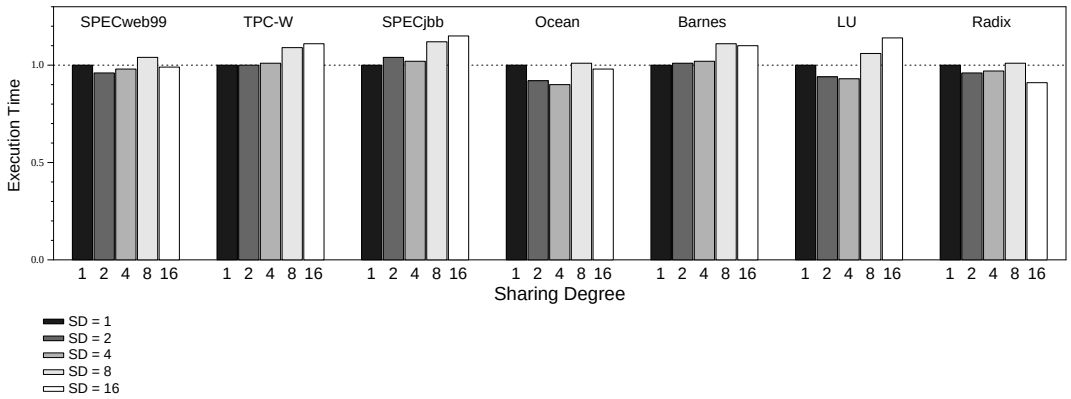


Figure 4.3: Normalized execution times with varying sharing degrees (normalized to SD=1)

Figure 4.3 shows the relative execution times for each application as the sharing degree is varied. As expected from Figure 4.2, SPECWeb99, Ocean, and LU have the best performance at sharing degree two or four. For Barnes and SPECjbb, increasing the sharing degree degrades performance, with the best performance at the sharing degree of one. The bar with the lowest average L1 miss penalty does not always correspond to the fastest execution time since each miss may affect the execution time with different weight (e.g., SPECWeb99 with SD=4). Critical misses that are not overlapped with other misses can increase execution times more than non-critical misses.

We draw three conclusions from these results. First, building high-degree shared caches for CMPs does not have any advantage in wire-delay dominated future technologies even when high degrees of application sharing exist. The increase in L2 hit latency in shared caches degrades performance more than the reduced misses improve it. Second, the sharing degree can change overall performance significantly. The difference between the best and worst sharing degree ranges from 9% to 22%.

Third, no single sharing degree provides the best performance for all the benchmarks. The best performing sharing degree varies across applications. Nevertheless, the SD=4 design point has the best average performance for the applications used in this evaluation, and would be the best compromise fixed design point for this mix of workloads for S-NUCA.

4.4 Dynamic Migration to Reduce L2 hit latencies

In the previous section, we showed the main drawback of higher sharing degrees is the increased hit latencies. In this section, we evaluate dynamic block migration to reduce hit latencies of shared caches. In the next section, we will evaluate another technique to mitigate the increased latencies.

Dynamic mapping (D-NUCA) addresses the problem faced by static mapping by allowing a block to go to multiple candidate banks, or a *bank set*. With proper placement and migration policies, D-NUCA enables the cache to place frequently accessed blocks in the banks closest to the CPU and less frequently accessed block in the banks that are farther away. Previous studies have shown that generational promotion that migrates blocks towards banks near the requesting processors yields good performance in uniprocessor systems [16, 53].

CMPs pose new challenges to dynamic mapping. First, migration in multiple directions can cause migration conflicts, with shared blocks ping-ponging between two processors. Second, searching blocks in bank sets becomes more complicated than single-processor D-NUCA organizations. Past studies have shown that centralized partial tags work well in uniprocessor D-NUCA [53]. In CMPs, however, besides increased bandwidth requirements on the central partial tag array, the cen-

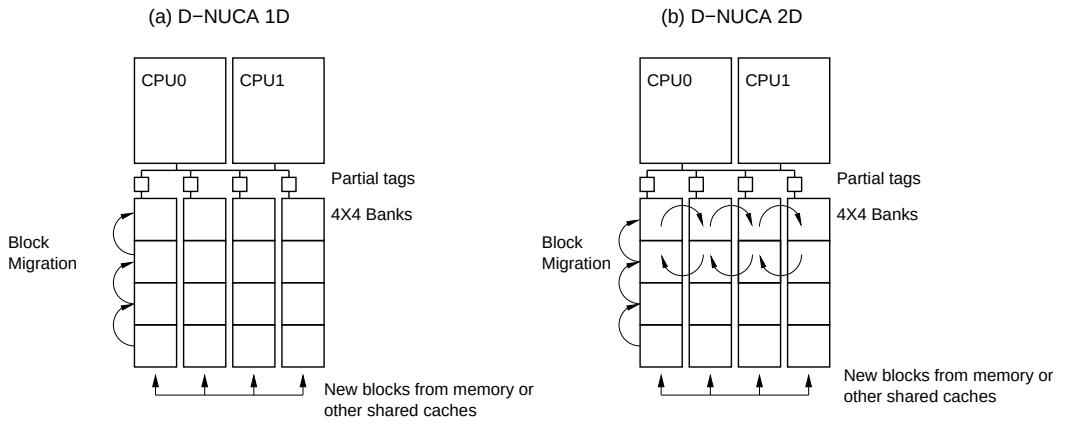


Figure 4.4: D-NUCA block migration policies (D-NUCA 1D and D-NUCA 2D)

tral partial tags cannot be located close to all processors, thus requiring multi-hop latencies to access the tags.

We address the challenge with two mechanisms: distributed partial tags and a dynamic lookup mechanism.

4.4.1 Migration Policies

A simple migration policy permits a block to be mapped to only one column and restricts migration to the vertical dimension only (D-NUCA 1D, Figure 4.4 (a)). In this policy, the vertical migration does not reduce the network latency for the horizontal traversal of requests and data blocks. We assumed the following bank set policies for misses: new cache blocks are inserted at the tail of a column bank set. For 8 and 16 sharing degrees, the victim banks are selected from the banks in the middle.

The second migration policy allows blocks to be mapped to any bank without restriction. Migration can happen in both vertical and horizontal directions

(D-NUCA 2D, Figure 4.4 (b)). This policy can further reduce hit latency by decreasing horizontal network latencies. However, it requires a more complicated search mechanism since an L1 miss might need to look up all banks in a shared cache. In D-NUCA 2D, new blocks are inserted at the column closest to the requesting processor.

Unlike single-processor D-NUCA caches, in which blocks are always promoted in one direction, multiple processors from different locations in CMPs can cause cache blocks to be promoted in conflicting directions. In a worst-case scenario, a block may ping-pong between two adjacent banks, with no reduction in hit latencies. To reduce unnecessary migration for conflicting promotion, we simulated two-bit saturated counters embedded in the cache tags, which allow a block to migrate only if the relevant counter for that moving direction is saturated.

4.4.2 Lookup Mechanisms

A partial tag structure replicates low-order bits of full cache tags as a way to reduce the number of requests to full tags [52]. In single-processor D-NUCA caches, centralized partial tags detect L2 misses early and reduce the number of requests to banks. However, the centralized partial tag approach has three drawbacks. First, since the global partial tag array is required to hold the information about all cache blocks in the cache, its access time and energy consumption will be relatively large. Second, since the centralized structure should be placed near the center of the cache, wire delays to and from it can be significant. Finally, the centralized tag array may require many ports since all primary cache misses must access it.

To overcome these drawbacks, we employ a distributed scheme: partial tags

are distributed over the columns, and each column's partial tag array tracks the state of blocks cached in that column. Any changes in a bank column's contents must be reflected in its partial tags synchronously. Cache lookups for a bank column always start from the distributed partial tag array. In the SD=8 or 16 cases, we replicate the partial tags at both ends of a column. This doubles the space overhead of partial tags, but greatly decreases the distance from processors to column partial tags.

In D-NUCA 1D, the search mechanism is straightforward with column partial tags. L1 misses are sent to the head of statically mapped columns, and the first bank and partial tags are accessed simultaneously. For a miss in the first bank, the column partial tags can command further lookups of other banks in the same column or start an L2 refill request. In D-NUCA 2D, the partial tags of all columns that a block can map to may be searched. L1 misses are first sent to the column closest to the requesting processor. If the block is not found in that column, other columns' partial tags are searched.

4.4.3 Results: Reducing Hit Latencies with Dynamic Mapping

In this section, we evaluate dynamic mapping policies which can potentially reduce long latencies with large sharing degrees. Performance improvements are achieved when the migration policy is successful and the reduction in latency dominates the increased latency of the more complex lookup mechanism. To isolate the effectiveness of dynamic migration from the overheads of the search mechanism, we evaluated two more configurations: perfect D-NUCA 1D and 2D caches. The two configurations assume an oracle searching mechanism that allows L1 misses to be

Sharing Degree	SD=1	SD=2	SD=4	SD=8	SD=16
S-NUCA	16.5	18.0	20.9	30.0	35.9
D-NUCA 1D Perfect	9.8	11.2	13.8	21.7	28.6
D-NUCA 1D Real	11.2	12.3	15.0	24.8	31.5
D-NUCA 2D Perfect	10.0	10.4	11.7	19.8	25.2
D-NUCA 2D Real	11.6	13.5	16.2	25.1	31.9

Table 4.4: Average D-NUCA L2 hit latencies with varying sharing degrees

sent directly to the L2 bank storing the requested block on a hit. L2 misses are detected without any overhead. The simulated system models other overheads such as network and bank bandwidth consumption for accesses and migration in detail.

Table 4.4 shows the average L2 hit times across all applications for five sharing degrees. With the perfect lookup mechanism, both 1D and 2D migration policies show significant reductions in L2 hit latencies. The latency reductions increase as the sharing degree increases. At SD=16, the perfect D-NUCA 1D and 2D policies reduce the average L2 hit latency by 22% and 33%, respectively compared to the S-NUCA design. However, with a realistic search mechanism with distributed partial tags, the hit latencies of D-NUCA are significantly increased from the perfect lookup mechanism, confirming the search mechanism is a key design issue with D-NUCA.

Figure 4.5 shows the relative execution times of the best performing sharing degree for the S-NUCA and D-NUCA design points across all applications. Each bar shows the SD with the best performance noted at the top. This figure illustrates two issues: (1) the performance potential of the two perfect search and migration mechanisms (1D and 2D perfect) and how closely the realistic implementations can match them, and (2) performance of two realistic D-NUCA designs compared to S-NUCA with the best sharing degree.

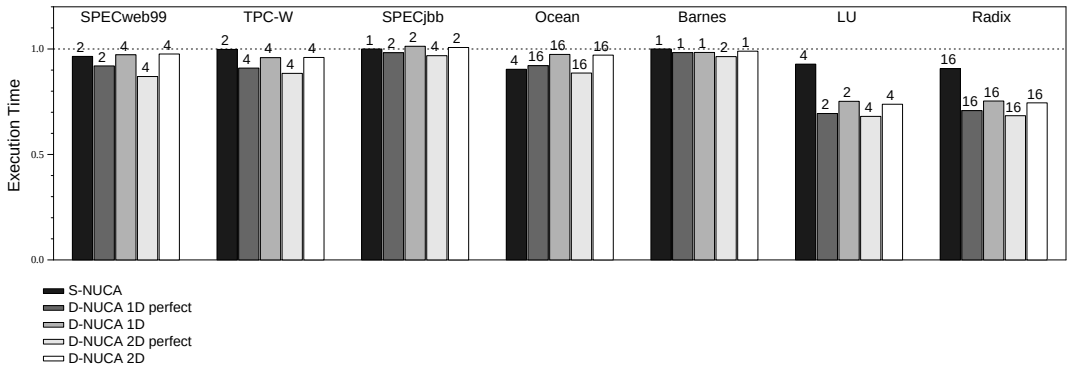


Figure 4.5: D-NUCA execution times (normalized to S-NUCA with SD=1)

The perfect search mechanisms with 1D and 2D dynamic migration can reduce execution time by 3-25%, except Ocean. For Ocean, although D-NUCA reduces average hit latencies, L2 miss rates are increased since blocks are not promoted quickly, and thus replaced by new blocks. For SPECjbb, the performance improvement is small, since SPECjbb does not take any advantage of the increased sharing degree, and the effect of dynamic migration is not high at low sharing degrees. With realistic search mechanisms, performance improvement of D-NUCA can be lost (SPECWeb99 and TPC-W). For LU and Radix, both 1D and 2D migrations show large improvement by 17%-20%. LU has a relatively large L1 data miss rate of 12%, but the entire working set nearly fits in the L2 caches. The reduction in L2 hit latencies directly improves performance. In Radix however, external memory accesses dominate performance due to both capacity and conflict misses. The increased bank associativity with dynamic migration reduced conflict misses significantly. Since shared caches, especially with high sharing degrees, are prone to conflict misses, the increased associativity helps avoid certain pathological conflict

Apps.	S-NUCA Best	D-NUCA 2D Best
SPECweb99	44	72 (40%)
TPC-W	25	34 (28%)
SPECjbb	4	6 (32%)
Ocean	10	18 (49%)
Barnes	12	15 (27%)
LU	31	60 (43%)
Radix	20	49 (48%)

Table 4.5: Bank accesses per 1K instructions

miss cases.

Although dynamic migration improves the performance of shared caches, the improvement is still modest (less than 5%) for 5 tested applications. Considering the complexity of a D-NUCA implementation and the extra energy consumption due to lookups, it is unlikely that implementing dynamic migration is justified for CMPs.

4.4.4 Results: Energy Trade-Offs

One concern with dynamic migration designs is the increased energy consumption due to the complex search mechanism and cache line movement. Instead of estimating the actual energy consumption, we indirectly show the total number of bank accesses of S-NUCA and 2D D-NUCA. For each application, we compared the best sharing degree of the two configurations in Table 4.5. The numbers in parentheses in the 2D D-NUCA row show the percentage of extra bank accesses compared to the S-NUCA case.

As expected, block migration increases the total bank accesses significantly. The extra bank accesses for block migration constitute 28-48% for the tested appli-

cations. In shared caches, unnecessary block migration may occur more frequently than private D-NUCA caches. Although we reduced the unnecessary migration by 2-bit saturating counters, the number of bank accesses due to migration is still significant.

D-NUCA lookup mechanisms also consume energy. For each L2 access, at least a part of distributed partial tags should be accessed. With D-NUCA 2D, the number of partial tag lookups may increase, if blocks are not found in the closest column.

4.5 Using L1 Prefetching to Hide L2 Hit Latencies

In this section, we investigate the effect of hardware-based strided prefetching [6, 47] on NUCA design alternatives. Since effective prefetching can tolerate L1 miss latencies, it can potentially diminish the effect of the increased L2 hit latency observed with larger sharing degrees. We evaluated the effect of strided prefetching on the CMP caches using an implementation similar to the one used by Beckmann and Wood [9], but restricted to L1 prefetching. The strided prefetching strategy uses three filters with 32 entries each to detect streams. The three filters, positive unit stride, negative unit stride, and non-unit stride use four consecutive misses before confirming a stream, and allocate an entry in an eight-entry stream table. As soon as a stream is detected, six consecutive prefetch requests are issued on behalf of the L1 caches. Prefetching stops when all MSHR entries are used or prefetches cross physical page boundaries. Prefetched cache blocks are stored directly into the caches, which may cause cache pollution problems. If the processors access a prefetched block, another prefetch for the stream is issued from the stream table if

	SPECweb99	TPC-W	SPECjbb	Ocean	Barnes	LU	Radix
L1 I Coverage	31.3%	14.1%	28.3%	20.3%	15.0%	14.8%	25.5%
L1 I Accuracy	46.4%	39.2%	34.0%	61.4%	50.4%	56.5%	49.4%
L1 D Coverage	14.0%	6.8%	0.5%	32.4%	12.2%	0.1%	10.3%
L1 D Accuracy	96.9%	90.0%	35.2%	95.3%	41.5%	0.1%	98.8%

Table 4.6: Prefetching coverage and accuracy for L1 instruction and data caches

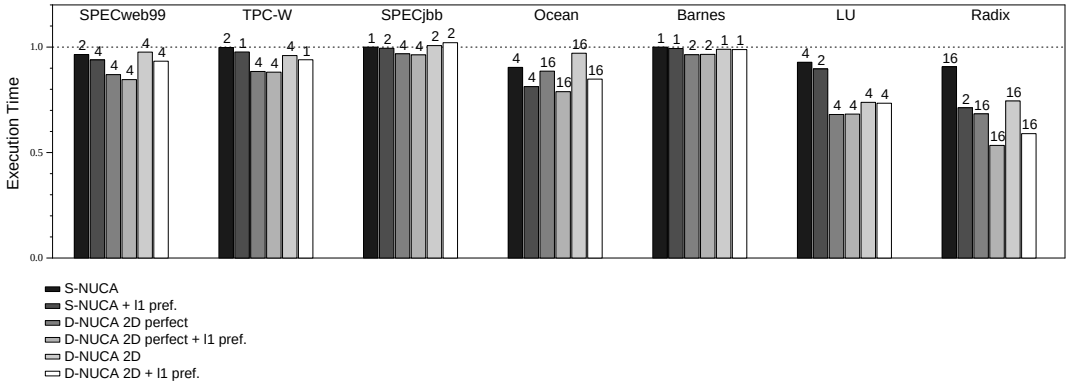


Figure 4.6: Execution times with L1 hardware prefetching (normalized to S-NUCA with SD=1)

the entry is still resident. Stream table entries are replaced using an LRU policy.

Table 4.6 shows the coverage and accuracy of L1 prefetching. *Coverage* is the ratio of prefetch hits to L1 misses, and *accuracy* is the ratio of prefetch hits to the total number of prefetches, ignoring any late prefetches that may partially hide latency. SPECjbb and LU have small coverage (0.5% for SPECjbb and 0.1% for LU) and relatively low accuracy. Prefetching is most effective for Ocean, with a coverage of 32% of the L1 data misses and 95% accuracy.

Figure 4.6 shows the relative execution times of S-NUCA and D-NUCA 2D with L1 prefetching compared to the baseline without prefetching using the best performing SD configuration for each run. For S-NUCA shared caches, L1 prefetching

can reduce execution time for SPECweb99 (3%), Ocean (10%), and Radix (20%). For SPECjbb, L1 prefetching does not reduce execution time due to the low coverage. Although prefetching can improve performance for many applications for the different configurations, it does not change the choice of the best average sharing degree for each design significantly. For six applications, the best sharing degrees with prefetching are either the same as or close to the best sharing degree without L1 prefetching.

We observe that prefetching can also improve dynamic migration. For applications where S-NUCA prefetching is effective, D-NUCA caches have similar performance improvements. This observation confirms that dynamic migration and prefetching are complementary memory latency reduction/tolerance techniques.

4.6 Per-Application Best Configuration and Per-line Sharing Degree

In this chapter, so far, we have showed which configurations of sharing degrees, block migration, and L1 prefetching have the best performance across entire benchmarks. In this section, we will discuss how per-application configurations can improve overall performance, dynamically adapting to the different usage patterns of applications.

We investigate the adaptability further into per-line sharing degrees. Even in an application, different memory areas may demand different sharing degrees. We show the possibility of having different sharing degrees for different regions of the same application.

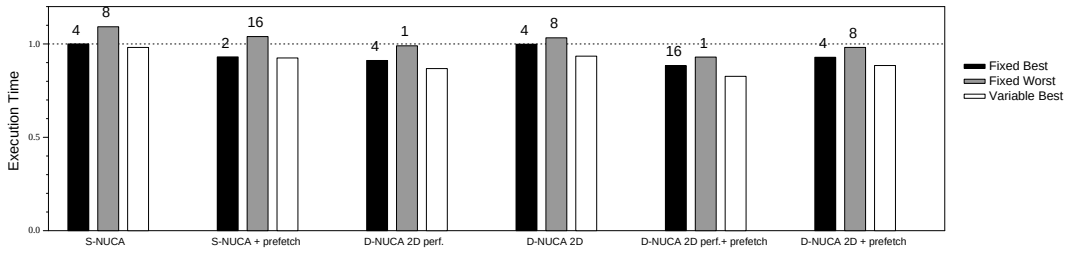


Figure 4.7: Execution time for fixed best, fixed worst and variable best sharing degree

	S-NUCA	S-NUCA + pref.	D-NUCA 2D
SPECWeb99	2 (1.4%)	4 (0.4%)	4 (0.0%)
TPC-W	2 (1.4%)	1 (0.4%)	4 (0.0%)
SPECjbb	1 (1.6%)	2 (0.0%)	2 (9.4%)
Ocean	4 (0.0%)	4 (2.4%)	16 (4.1%)
Barnes	1 (2.0%)	1 (0.9%)	1 (3.6%)
LU	4 (0.0%)	2 (0.0%)	4 (0.0%)
Radix	16 (6.6%)	2 (0.0%)	16 (25.4%)
Fixed Best	4	2	4

Table 4.7: Per-application best sharing degrees

4.6.1 Per-application Best Configuration

Since the underlying cache framework permits different degrees of sharing on the same hardware, further opportunity exists: The cache can be configured differently to have the ideal sharing degree for each specific application or for individual cache lines. Figure 4.7 shows a comparison of the average execution time across all applications for the S-NUCA and D-NUCA designs normalized to the S-NUCA design with the best sharing degree of four. For each policy, we show the performance with the best fixed sharing degree across all applications, the worst fixed sharing degree, and a per-application "variable" degree. Choosing the best sharing degree

at a finer granularity provides a small but measurable (5%) speedup for the more aggressive policies. In Table 4.7, we list the ideal per-application sharing degrees for each policy and show the percentage speedup over the best fixed sharing degree for that policy. For D-NUCA, SPECjbb, Ocean, and Radix showed a measurable boost of 5-25% from fixed best sharing degrees, by using per-application sharing degrees.

4.6.2 Per-line Sharing Degree

Sharing degree can affect individual cache blocks in different ways based on the sharing patterns of the block. For private blocks, which will never be replicated in other caches, a low sharing degree can reduce the access latencies without hurting caching efficiency. For shared blocks, a higher sharing degree may be more beneficial than lower sharing degree when the reduction of replicated blocks can decrease miss rates significantly. In this section, we investigate the potential benefit of multiple sharing degrees for different classes of blocks.

We divide the address space into private and shared block addresses, and assign different sharing degrees to the two address classes. To evaluate the per-line sharing degrees, we simulated S-NUCA with the two different sharing degrees, *private* and *shared* (PSD and SSD). Since our full-system simulator does not have support for distinguishing private and shared blocks, we used an approximate method of tracking access patterns for the entire address space during run time. Until more than one processor accesses a block address, we assume the address is a private block. Once an address is recognized as a shared block, the block is re-mapped to caches by the default sharing degree. We assumed that the sharing degree is always higher than or equal to the private sharing degree.

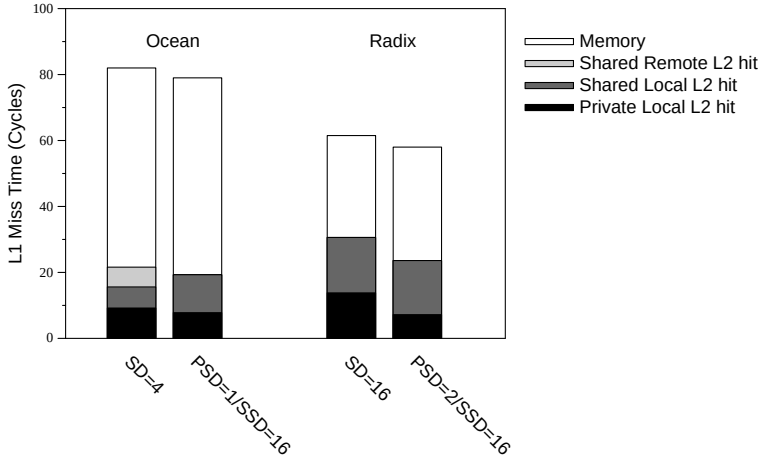


Figure 4.8: L1 miss latency decomposition with per-line sharing degrees

Figure 4.8 presents the breakdown of L1 miss penalties for the two applications in which per-line sharing degree is effective. We measured all combinations of private and shared sharing degrees. The two bars for each application represent the baseline ideal uniform sharing degree and the best per-line sharing degree configuration. We divided the local L2 hit latencies into private and shared accesses.

For the two applications, per-line sharing degree reduced execution time by 7% and 6%. With fixed sharing degree, the best sharing degree was 4 (Ocean) or 16 (Radix). When different sharing degrees are allowed for private and shared blocks, the best combination is 1 or 2 for private sharing degree and 16 for shared sharing degree. For these two benchmarks, access latencies to private blocks are reduced by having lower sharing degrees (1 or 2), while replications are minimized with high shared a sharing degree (16). With a low private sharing degree, private blocks, which will never be accessed by other processors, are placed on a small range of banks close to processors. With a high shared sharing degree, block replication is reduced, although the shared blocks may be spread across a larger range of banks

than private blocks, thus increasing access latencies. It is possible that a finer-grained distribution of lines to banks would improve performance.

4.7 Summary

The CMP organization that we introduced in this chapter is designed to support both low-latency, private logical caches as well as highly shared caches, simply by adjusting the mapping of the same address on different processors to the L2 cache. The results showed that–compared to private, non-shared L2 partitions–the L2 latency more than doubled for a fully shared cache. The results also showed that the fully shared cache could eliminate a third of off-chip misses. Clearly, a large opportunity exists if this gap can be bridged.

The S-NUCA organization (static mapping) worked best with a low-to-medium sharing degree for all applications; the extra hit latency was simply too detrimental with larger sharing degrees. Consequently, we evaluated L1 prefetching and dynamic migration of lines, attempting to reduce the average hit latencies and make the larger sharing degrees more effective. L1 prefetching worked uniformly well, but did not drive the ideal sharing degree significantly in one direction or the other, even though the L1 miss rates were reduced.

Dynamic migration (D-NUCA 1D and 2D) showed modest performance improvements, despite reductions in average hit latency. However, for a subset of applications, D-NUCA drove the ideal sharing degree to higher sharing degrees, showing that mechanisms to reduce latency can indeed make higher-degree shared caches the optimal point. It remains to be seen whether the added complexity and power consumption justify moving to a D-NUCA design since only a subset of the

applications benefit appreciably; we think it unlikely to be justifiable.

Probably the best: an S-NUCA organization with a sharing degree of two or four. However, the D-NUCA results still hold promise, and we are continuing to explore ways to exploit the flexible mapping. Treating different classes of lines with different sharing degrees showed significant potential for two applications.

In this chapter, we investigated cache sharing as a miss reduction technique for CMPs. We showed that shared caches can reduce external memory accesses significantly and proposed techniques to overcome the latency disadvantage of shared caches. However, the techniques in this chapter do not address coherence communication in large-scale systems beyond the on-chip shared caches. In the next chapter, we will propose a technique, which uses speculation to address the communication latency aspect of MP cache performance.

Chapter 5

Coherence Decoupling: Using Speculation to Hide Coherence Latency

In this chapter, we use the speculative execution capability of microprocessors to hide long communication latencies in multiprocessors. In the past two decades, techniques based on speculation have been used to improve microprocessor performance. With speculation, microprocessors can continue to execute instructions with predicted results. Rather than incurring the latency of waiting for the outcome of an event, the outcome is predicted, allowing execution to proceed with the prediction. The prediction is verified when the outcome of the event is known, and corrective action is taken if the prediction was wrong. Speculative execution has been successfully used to overcome performance hurdles in a variety of scenarios, for example, branch instructions (control speculation) [86, 98], ambiguous dependences

(dependence speculation) [76], parallelization (speculative parallelization) [99], and locking overheads (speculative lock elision) [88].

In shared memory multiprocessors, communication misses occur when other processors update shared data and the blocks in local caches become invalid. Subsequent accesses to the invalid data in the local cache will incur a cache miss (communication miss or coherence miss). In the conventional cache coherence protocols, such cache misses block the execution of instructions dependent upon the missed data, incurring performance losses. By using speculation, the coherence-missed data can be predicted, and the processor can continue to execute the dependent instructions. If the missed communication data can be predicted correctly, speculation can reduce or completely hide the long latencies to resolve communication misses.

We propose a technique called *coherence decoupling*, which applies speculation to the problem of long-latency shared-memory communication. This technique reduces the effect of these latencies, but neither exacerbates the programmer's task nor makes correctness of the coherence protocol more difficult to ascertain. Coherence decoupling breaks the communication of a shared value into two constituent parts: (i) the acquisition and use of the value, and (ii) the communication of the coherence permissions that indicate the correctness of the value and thus the execution. In traditional cache coherence protocols, these two aspects of communication have been merged into a single protocol; obtaining the coherence permissions must strictly occur before use of the data, thus serializing the two. Coherence decoupling enables separate protocols for the speculative use and eventual verification of the data. A *Speculative Cache Lookup* (or SCL) protocol provides a speculative value as quickly as possible, while in parallel the *coherence protocol* executes and eventually

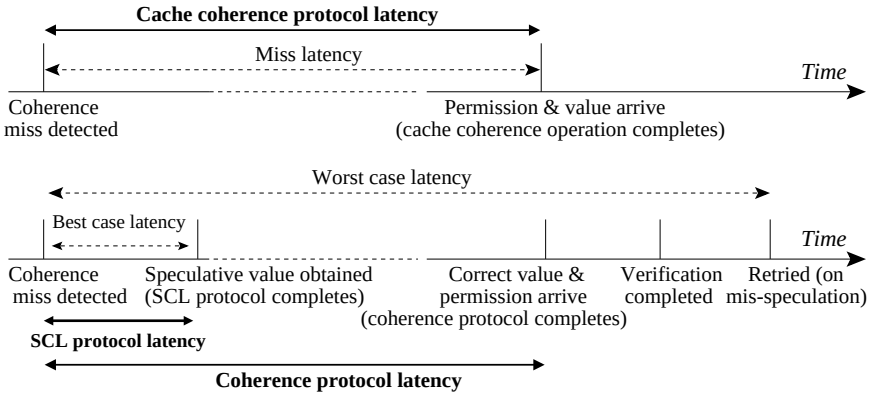


Figure 5.1: Timing diagram for coherence decoupling

produces the correct value along with the requisite access permissions.

Separating the SCL protocol and the coherence protocol allows each to be tuned independently. This capability enables novel optimizations that permit higher performance with less complexity than traditional protocol optimizations. The separation also allows the two to be overlapped. The top part of Figure 5.1 shows the timing of events, with a conventional coherence protocol, for a read to a cache line that requires a change in coherence permissions. The cache coherence operation is followed by the arrival of the line with the correct data and the appropriate permissions, at which point the data can be used. The bottom part of Figure 5.1 shows the timing of events in a system with coherence decoupling. Here the SCL protocol could speculatively return the data earlier, for example, if a tag match occurs in a local cache (even if the line is invalid), while simultaneously launching the invalid-to-shared upgrade via the coherence protocol. When the coherence protocol returns the permissions and the correct value, the value is compared to the value returned by the SCL protocol. If the values are identical, the speculation was correct, and the coherence latency will have been partially or fully overlapped with useful com-

putation ("best case" in the figure). If the SCL and coherence protocol values differ, a full or partial rollback must occur, resulting in a performance loss compared to no speculation ("worst case" in the figure). The utility of coherence decoupling, as with all speculation policies, depends on the ratio of correct to incorrect speculations, the benefit of a successful speculation, and the cost of recovery.

5.1 Related Work

The prior research in reducing multiprocessor communication overhead that is most relevant to coherence decoupling falls into three broad categories: (1) customized coherence protocols, (2) speculative coherence operations, and (3) speculation on the outcome of events in a multiprocessor execution. We describe each category in turn below.

Customized coherence protocols attempt to specialize underlying coherence protocols to reduce communication and coherence latencies for special cases. The Stanford Dash multiprocessor [61] included directory protocol optimizations for specific sharing patterns, as did the pairwise sharing protocol in the Scalable Coherent Interface [45], and migratory sharing protocols [18, 101, 21]. Some protocols were proposed to adapt to different sharing patterns [108, 22], or traded-off write invalidate and write-update protocols [5, 77, 90].

Another set of protocols exposed coherence protocols to software; cooperative shared memory [42] used software directives to allow applications to guide the coherence protocols with check-in and check-out primitives. Exposure to software reached its zenith with the Stanford Flash [55] and Wisconsin Tempest/Typhoon [91], which enabled software to customize coherence protocols on a per-application basis [26].

These research directions were discontinued as it became apparent that protocol customization was too difficult for most programmers.

Speculative coherence operations are in some sense the converse of the coherence decoupling approach. Coherence decoupling performs speculative *computation* to tolerate the latencies of unmodified coherence protocols, using a SCL protocol to increase speculation accuracy without complicating the base coherence protocol. Speculative coherence operations, conversely, perform no actual speculative computation, but speculatively initiate coherence messages (e.g., invalidates or upgrades) in the base protocol to reduce the latency eventually seen by mandatory coherence operations. Lebeck and Wood proposed Dynamic Self Invalidation [58]: processors speculatively flush their blocks based on access history, reducing the latency of later invalidations by remote writers. Subsequently, Mukherjee and Hill proposed a "coherence message predictor" [79] that initiated coherence messages speculatively, triggered by other messages indexing into cache block indexed two-level adaptive predictors [110]. Kaxiras and Goodman evaluated message predictors using PC-indexed tables rather than address-indexed tables [49]. Lai and Falsafi augmented the use of tables containing patterns of messages by restricting these tables to memory demand requests only [56], showing that limiting the table to demand request messages only provided a more effective predicted stream of coherent block read requests. They also replaced the access counts used in Dynamic Self-Invalidation with two-level adaptive prediction in a "last-touch predictor" [57]. Kaxiras and Young explored a range of policies to predict the set of sharers of a cache line [50], as did Martin et al. [66] with "destination set prediction," which they used to enable more complex, adaptive coherence protocols that exploited the predicted sets of sharers.

The prior work most similar to coherence decoupling is the work that speculates on the outcome of a multiprocessor event — a form of value speculation different from the typical value prediction strategies. There are two categories of such techniques. The first category is *speculative synchronization*, where the outcome of a synchronization event is speculated. For example, a lock is speculated to be unheld, permitting entry into critical sections [88, 89, 71]. The similarity with coherence decoupling is that both techniques employ speculative access to shared variables, which for speculative synchronization is limited to locks. Temporally silent stores and the MESTI protocol [63], a proposed alternative to speculative synchronization, is, in our classification, an example of a customized coherence protocol. The MESTI protocol exploits the predictable behavior of the values of lock variables to reduce the coherence protocol overhead in a lock handoff, but is neither a speculative protocol, nor does it launch speculative operations.

The second category of event outcome speculation techniques use speculation to overcome the performance limitations of strong memory models [31, 109, 84, 34]. These techniques speculate that a memory model (e.g., sequential consistency) will not be violated if memory references are executed in an optimistic fashion. Memory operations that have been carried out optimistically are buffered and these buffers are checked to see if the optimistic execution has resulted in a possible violation of the memory consistency model [32]. Execution is rolled back in case of a violation. This form of speculative execution is widely used in commercial multiprocessors today.

Recent work by Martin, et al. [67], proposes token coherence, a new way of implementing cache coherence in a multiprocessor with an arbitrary interconnec-

tion network. Token coherence distinguishes between a performance protocol and a correctness protocol. A correctness protocol returns the correct value in all cases; a faster performance protocol returns the correct value in most cases. Unlike coherence decoupling, the performance protocol, however, always returns a correct, non-speculative value, and token coherence is not a technique that employs speculative execution.

5.2 Potential Latency Reduction with Coherence Decoupling

In this section, we categorize L2 cache misses in multiprocessors, and show the potential communication misses to be hidden by coherence decoupling. In Section 5.2.1, we discuss how L2 misses are classified, and which classes of misses can be effectively hidden with coherence decoupling. In section 5.2.2, through experimental simulations, we show what fraction of misses can potentially be covered with coherence decoupling.

5.2.1 Classification of Communication Misses

In uniprocessor caches, cache misses are classified to capacity, conflict, and cold misses [43]. Besides the three types of misses, communication in shared memory multiprocessors with invalidation-based coherence, adds one more type of misses, coherence misses. Coherence misses, or communication misses, occur when blocks in local caches are invalidated by other processors which intend to update the blocks. Subsequent accesses to the local copies cause cache misses, since the blocks are in invalid states. Such coherence misses can only happen with multiple processors

sharing cache blocks. Coherence decoupling reduces the effect of long latencies for such coherence misses.

Coherence misses can be further divided into three classes, false sharing, true sharing, and silent stores, and the effect of coherence decoupling can vary for different classes.

- False sharing: False sharing misses occur due to block-based management of coherence [25]. A cache block is the atomic unit of coherence in cache coherence protocols. Even if processors update non-overlapped portions within a block, any write to the block should invalidate the copies of the same block in other processors. If a block is invalidated to update a word in the block, subsequent reads to a different word in the same block become false sharing misses.

  For false sharing misses, the portion of the cache block that a processor is accessing in its local cache has not been updated by other processors, even if the states of blocks are invalid. Therefore, predicting values for false sharing misses is easy by using invalid values in the local cache, which have by definition not been updated by remote processors.

- Silent stores: Silent stores differ from false sharing in that the data portion is actually changed by other processors. As shown in Lepak et al [62], however, the new value other processors write to the block is the same as the old value. Temporal silent stores are slightly different from plain silent stores [63]. In temporal silent stores, the old value in the local cache is updated with a different new value, but eventually, the value is changed back to the original one, before the local cache is accessed. In both cases, the old data value in the local cache is correct.

- True sharing: We define true sharing only for the coherence misses that are neither false sharing misses nor silent stores. The data values in the local caches have been changed and furthermore, new values are different from old ones. For true sharing misses, using the invalid value in the local caches will cause a miss prediction and recovery.

5.2.2 Miss Profiling Results

Figure 5.2 shows a breakdown of L2 read misses for a number of cache configurations (1MB and 4MB capacity, and 128-byte and 512-byte cache lines). The number of processors for the experiments is fixed to 16. We partition read misses into coherence misses and "other" misses, which include capacity, conflict, and compulsory misses.

We define coherence misses as having a matching tag in the L2 cache but the wrong coherence state (e.g., invalid state on a read), thus requiring remote communication. The actual coherence misses can be more than what are measured in the experiments, since invalidated blocks can be replaced in local caches. However, this restricted definition can show conservatively the potential of coherence decoupling.

We subdivide coherence misses into false sharing, silent stores, and true sharing misses. Coherence misses are counted as true sharing misses only when the correct values differ from those in the local cache's stale copy. Silent stores update a cache block, but the values have not been changed from the old values stored in the local invalid block. The silent store bar in the figure includes both temporally silent stores and silent stores. For false sharing misses and silent stores, local cache lines in invalid state will have the correct values. The sum of silent stores and true sharing misses are the traditional address-based true sharing misses by Dubois'

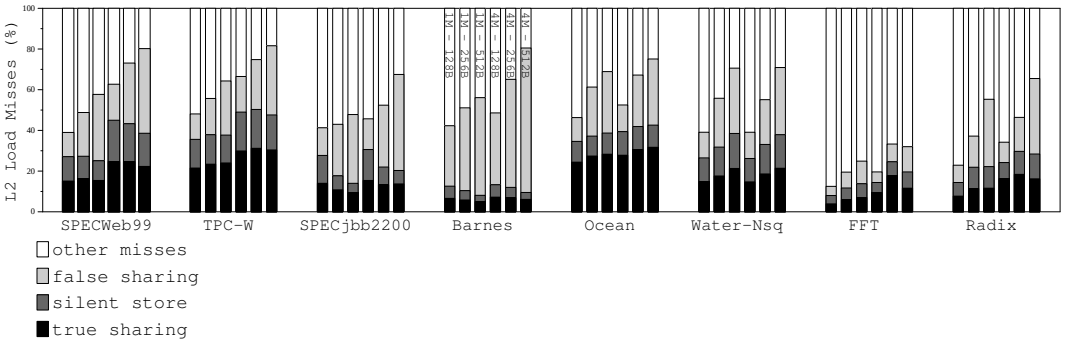


Figure 5.2: L2 load miss breakdowns (false sharing, silent store, and true sharing)

definition [25].

The data show that the fraction of coherence misses is significant for every one of the commercial benchmarks (a minimum of 12% in FFT). The data corroborate the expected trend that as the cache size grows — for a fixed size workload — the coherence misses increase, to 80% in SPECWeb, 81% in TPC-W, and 67% in SPECJbb. Given the enormous cache sizes in future server-class systems (36MB per processor die in IBM's Power5 system), we expect that coherence misses will be a significant and growing component of communication in future multiprocessor systems.

A significant fraction of L2 load misses across the benchmarks are coherence misses caused by silent stores, from 7% (Barnes) to 16% (Ocean). The base `CD` protocol will predict the correct value for the silent stores as well as for false sharing misses. The ratio of false sharing misses (plus silent stores) to true sharing misses increases as the block sizes increase, for all benchmarks. Of those being simulated, the cache configuration with the lowest average miss rate (4MB with 512B blocks) shows that from 13% (FFT) to 71% (Barnes) of all L2 misses result from false

sharing. If cache size growth outstrips working set size growth, as is certain for some benchmarks, coherence misses in general and false sharing in particular will increase as a fraction of L2 misses.

5.3 Coherence Decoupling Architecture

Coherence decoupling separates a cache coherence protocol into two parts: (i) a speculative cache lookup (SCL) protocol, which returns a speculative value that can be used for further computation, and (ii) a coherence protocol, which returns the correct value (as defined by the memory consistency model) and the requisite permissions to use the value. If the SCL protocol can return a value faster than the coherence protocol, the computation using the value and the coherence operations can be overlapped. Higher accuracy in the SCL protocol allows for more frequent hiding of coherence protocol latencies, allowing simpler but lower performance coherence protocols to be used without a commensurate performance penalty. In this section, we consider three components to support coherence decoupling (Section 5.3.1), and discuss how to ensure the correctness of coherence decoupling (Section 5.4).

5.3.1 Three Mechanisms to Support Coherence Decoupling

Coherence Decoupling system architectures must provide three mechanisms: i) a *value acquisition mechanism* to obtain speculative load values, ii) a *verification mechanism* to receive a correct value and permission by communicating with other processors, iii) a *recovery mechanism* to detect misspeculation and recover correctly from it.

- Value Acquisition (SCL protocol): The value acquisition must provide load values with high accuracy. There are three criteria a good value acquisition protocol should meet. i) The latency of obtaining a value is important to minimize processor stalls. It must return a speculative load value far ahead before a coherence protocol returns a correct value. ii) A good value acquisition protocol must be able to generate speculative values for the majority of coherence misses. iii) The accuracy of the protocol is also important to minimize the recovery cost of misspeculation. We use the Speculative Cache Lookup (SCL) protocol as our value acquisition protocol. The most basic protocol uses values in the local cache, although the cachelines are in the invalid state. This basic protocol attempts to eliminate the effect of false sharing, silent stores [62], and temporal silent stores [63].

- Verification: Any coherence protocol can be used for verification. This safety net can take much longer than the SCL protocol, but must always return a correct value. This verification protocol must also ensure global consistency in multiprocessors. In the discussion, we use an invalidation-based protocol with a snooping bus.

- Recovery: A recovery mechanism recovers from an inconsistent state incurred by misspeculation. Any misspeculation recovery mechanism in modern speculative processors can be used. For example, the recovery mechanisms from branch misspredict, or from value mispredication can be used. However, the recovery for coherence decoupling may require the ability to buffer deep speculative execution, since the latency of a coherence protocol is already hundreds of processor cycles, and it will grow. The other requirement for the recovery

mechanism is to recover as quickly as possible. In this work, I model *flushing*, which squashes all the instructions succeeding the misspeculated loads.

To support coherence decoupling the system architecture must: (i) *split*, providing a means to split a memory operation into a speculative load operation and a coherence operation, (ii) *compute*, providing mechanisms to support execution with speculative values, and (iii) *recover*, providing a means for detecting a mis-speculation and recovering correctly from it.

Splitting a memory operation (*i* above) into two sub-operations is straightforward, as is the recovery process (*iii* above) of comparing the results of the speculative load operation and the coherence operation to detect a mis-speculation. The speculated value may be buffered in an MSHR, which then compares the value against the correct value when the coherence protocol returns the cache line.

To support speculative computation (*ii* above), the same mechanisms that are used to support other forms of speculative execution can be used. Since coherence latencies are growing to hundreds of cycles, however, current microarchitectural mechanisms to support in-processor speculation (e.g., branch speculation) are likely to be inadequate. Mechanisms that can buffer speculative state across hundreds to thousands of speculative instructions will be necessary.

In this chapter, for recovering from mis-speculations, we model the standard recovery policy for techniques that use deep speculation: squashing the offending instruction and all succeeding instructions.

5.3.2 Correctness of Coherence Decoupling

Coherence decoupling is a form of value prediction for multiprocessors. As Martin et al. have observed [69], implementing value speculation correctly requires hardware that performs the same function as that used for aggressive implementations of sequential consistency (SC) and vice versa.

Coherence decoupling relies upon the above observation for correctness. Obtaining a value speculatively with an SCL protocol — and later verifying the speculation via the coherence protocol — is analogous to carrying out a memory operation speculatively assuming that the memory consistency model will not be violated, and using the coherence protocol to verify the speculation. Thus, if we use the same hardware to implement coherence decoupling that we use to implement aggressive implementations of SC, coherence decoupling can be implemented without any correctness implications for the memory consistency model.

5.4 SCL Protocols for Coherence Decoupling

A wide range of SCL protocols for coherence decoupling are possible. Although the SCL protocols can be combined with arbitrarily-complex coherence protocols, coherence decoupling enables these aggressive SCL protocols to be backed by a simple, easily-verifiable coherence protocol. In this work we therefore measure only a simple invalidation-based coherence protocol and rely on the SCL protocols to improve performance.

An SCL protocol has two components. The first is the *read component* — the policy for obtaining the speculative value (i.e., where the protocol searches for

SCL Component	Policy	Description
Read	`CD`	Use the locally cached incoherent value for every L2 miss
Read	`CD-F`	Add a PC-indexed confidence predictor to filter speculations
Update	`CD-IA`	Use invalidation piggyback to update all invalid blocks
Update	`CD-C`	Use invalidation piggyback if the value is special (compressed)
Update	`CD-N`	Update all sharers after N writes to a block (N=5 in Section 6.4)
Update	`CD-W`	(Ideal): Update on every write if any sharers exist

Table 5.1: Coherence Decoupling protocol components (read and update)

a speculative value to use). The second is the *update component*, in which the SCL protocol may speculatively send writes to invalid cache lines (former sharers) to increase the probability that a subsequent coherence decoupled access will read the correct value. This component trades increased bandwidth — consumed by sending speculative writes around the system — for improved speculation accuracy. The update component may be null in some SCL protocols.

5.4.1 SCL Protocol Read Component

The first policy for the read component we propose simply uses the value in the local cache if the block is present (i.e., the tag matches) and if the block is either in an invalid state. We call this `CD`, for basic coherence decoupling.

Since `CD` speculates on the value of every load operation that finds a matching tag (but with the wrong permission), it may incur a large number of mis-speculations, triggering too many rollbacks. The next SCL read component policy we propose, called "Coherence Decoupling + Filter" (or `CD-F`), employs a confidence mechanism — a PC-indexed table of counters — to throttle speculations. For some extra hardware, `CD-F` reduces the number of times speculation is employed (i.e., it reduces the *coverage*), thereby decreasing the total number of mis-speculations, but

improving the average speculation *accuracy* over the base `CD` protocol.

In general the read component of an SCL protocol could return a (possibly incorrect) value from anywhere it finds in the system, if the latency of doing so is sufficiently lower than the latency of accessing it through the coherence protocol. In a directory-based cache coherent machine, for example, the SCL protocol could first access the local cache and then the home memory of the invalid line, using the invalid data while the home directory communicated with an exclusive owner of the block. In another example, the value could reside in a geographically-proximate cache in a hierarchical multiprocessor (e.g., another cache on the same chip in a multiprocessor built from CMPs). In this dissertation, however, we consider only a flat symmetric multiprocessor leaving the issue of SCL protocols for hierarchical systems to future work.

An SCL protocol with only a read component (and a null update component) speculates correctly if the contents of the accessed word in the invalid block have not changed remotely since being invalidated (false sharing [25]), have been changed remotely to the same value (silent stores [62]), or have been changed remotely to a different value and then changed back to the original value (temporally silent stores [63]). This capability allows the problem of false sharing to be greatly mitigated. As long as there is sufficient work for the processor to do after it speculates on falsely shared data, the coherence protocol latency for such a request can be overlapped completely. A successful `CD` protocol will prevent the programmer from having to recode data structures to reduce false sharing (if they can even figure out that false sharing is occurring in the first place).

5.4.2 SCL Protocol Update Component

We can further attempt to improve the accuracy of speculation for truly-shared data by adding *update components* to the SCL protocol. An update component speculatively sends updated data around the system and writes them into invalid cache lines. The update component of an SCL protocol thus trades increased speculation accuracy for the extra bandwidth consumed by the updates.

A variety of protocols for the update component of an SCL protocol, with different accuracies and bandwidth requirements, are possible. We present several such protocols in this section; it is easiest to view them as variants of a basic write-update protocol. It is important to note that since these updates are speculative, they can be completely non-blocking for the writer and can proceed in parallel with other operations. If a speculative write finds a copy of the line which is not in invalid state, the write is simply dropped and correctness preserved. This capability is in contrast to a canonical write-update cache coherence protocol which requires the writer to view the transmission of the write updates as a blocking operation.

Our first update component for an SCL protocol, `CD-IA`, piggybacks the value created by the writer along with the invalidation message used to invalidate remote caches. The message size is increased to include a data packet in addition to the address packet. However, since we model a bus-based broadcast coherence protocol in this dissertation, `CD-IA` updates the data in all caches which have the block (i.e., caches already in an invalid state) and not only the sharing caches that need to be invalidated.

`CD-C` is a variant of `CD-IA`; it uses compressed updates to reduce the message overhead. For the commercial workloads studied in this dissertation, many of writes

that result in an invalidation message frequently write the values 0, 1, or -1. `CD-C` piggybacks updates for only these values to the initial invalidation message, allowing these updates to be communicated to remote caches by adding only two additional bits to the invalidation message.

The remaining protocols for the SCL update component that we consider also send updates after the initial invalidations have been sent. Consequently these additional updates require additional messages. `CD-N` broadcasts the dirty line after N updates have been made by the same writer. Other possible policies might broadcast the block after *every* N writes, or broadcast the block after the (predicted) last write to the block. With the bus-based interconnect that we model, which has limited bandwidth, these policies performed much worse than the others. They may be more compelling on higher-bandwidth topologies, which we leave to future work.

Finally, `CD-W` is an ideal policy that sends an update on every write, if invalid sharers exist. That is, it uses a conventional write-broadcast protocol for the update component of the SCL protocol. In a machine with directory-based cache coherence, the writer could maintain the list of sharers after invalidations for propagating occasional writes, or the system could use destination set prediction for guessing which nodes hold invalid copies of a line [50, 66]. Table 5.1 summarizes the SCL protocol components that we consider in this dissertation. Note that the read component of the SCL protocol is orthogonal to the update component of the SCL protocol. Thus either of the read components (`CD` or `CD-F`) could be used in conjunction with any of the update components (`CD-IA`, `CD-C`, `CD-N`, or `CD-W`), or even with a null update component. To reduce the number of combinations, however, for the remainder of this chapter when we discuss an SCL protocol with a non-null update component

(CD-IA, CD-C, CD-N, or CD-W), we will assume that it uses CD for its read component.

Clearly other options for speculatively passing around data are possible, trading off speculation accuracy with message bandwidth. Existing cache coherence protocol optimizations for *correctly* passing data can be leveraged into have *speculative* versions. For example, we could have speculative competitive write-update protocols, or speculative customized protocols that can dynamically learn the communication pattern of an application and try to optimize the data communication. Such protocols are left for future work.

5.5 Evaluating Coherence Decoupling

We ran the coherence decoupling experiments on *MP-Sauce*. Network contention due to speculative updates is also modeled (except for the ideal CD-W protocol). Table 5.2 lists the most relevant machine parameters from the simulated system.

We simulated three commercial applications and five scientific shared-memory benchmarks from the SPLASH2 suites. The three commercial workloads are TPC-W using a MySQL backend running on Apache, SPECWeb99 running on Apache, and SPECJbb using the IBM Java virtual machine. The SPLASH applications we simulate are Barnes, Ocean, Water-nsq, FFT, and Radix.

Since multi-threaded, full-system simulations produce results that vary from run to run, we replicated the methodology in other studies and ran each experiment multiple times, injecting small timing variations [3]. We report the mean of the execution time across the multiple runs as our experimental result.

In this dissertation, we limit our simulations to 16-node SMP systems, for two reasons. First, although small-scale hierarchical (CMP-based) NUMA systems

Feature	Parameters
Issue width	4
Window size	512-entry RUU
Number of CPUs	16
L1 cache	split I/D, 128K, 4-way, 128-byte block
L2 cache	unified, 4M, 4-way, 128-byte block
MSHR size	32
Base protocol	bus-based MOESI
Bus bandwidth	12.8GB/s
L1/L2 hit latencies	2 cycles / 24 cycles
Memory access latency	460 cycles
Cache-to-cache latency	400 cycles

Table 5.2: Simulated system configuration for Coherence Decoupling

provide opportunities for coherence decoupling, they are not yet prevalent. Second, more traditional, directory-based CC-NUMA multiprocessors are typically too large to simulate in our full-system environment—the AIX 4.3.1 operating system that we simulate can support configurations only up to 24 processors. We expect that the relative performance benefits we show will only increase in larger-scale systems, where coherence misses are more frequent and latencies are longer.

5.5.1 Microbenchmarks

To understand the effectiveness of coherence decoupling, we show the results of two simple microbenchmarks, which are designed to generate false sharing misses. `simple-fs` loads falsely shared data, while executing both dependent and independent instructions every loop iteration. The ratio of dependent to independent instructions is set to 1:3. The dependent instructions are simple additions and multiplications, which use the value returned from a load that incurred a false sharing miss. `critical-fs` generates a false sharing miss on each iteration, but calculates

```
for(i = 0; i < MAX; i++) {
   fval = array[i].value;  /* false sharing miss */
   dep_val1 = fval + 2;
   indep_val1 = local + 2;
   dep_val2 = fval * 3;
          …
   /* dependent and independent instructions */
}
```

(a) simple-fs

```
for(i = 0; i < MAX; i++) {
   index = array[index].value; /* false sharing miss */
   sum += index;
}
```

(b) critical-fs

Figure 5.3: Microbenchmark codes

the address of that load using the value returned from the false sharing miss of the previous loop iteration. Figure 5.4 presents the key microbenchmark code fragments on the left half of the figure.

The right half of Figure 5.4 shows the microbenchmarks' normalized execution times, varying cache-to-cache communication latencies from 200 to 1000 cycles, with each microbenchmark using both the baseline and the `CD` protocol. `simple-fs` has speedups from 12% to 17% over the base case, as communication latencies increase. With a 512-entry RUU, `CD` can execute approximately 120 dependent instructions, none of which can be executed in the baseline system until the falsely shared data return. Despite this additional latency tolerance, however, `CD` can not hide communication latencies past a certain size due to the finite instruction window size (512 entries), after which point performance degrades more quickly as latencies increase.

`critical-fs` forces a data dependence between two loads, placing consecu-

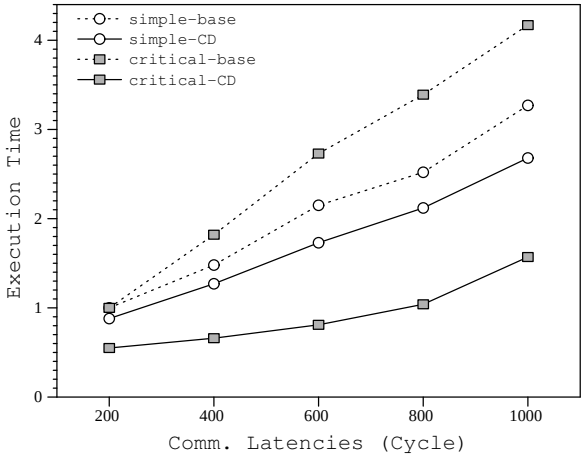


Figure 5.4: Microbenchmark performance results

tive false sharing misses on the critical path of execution. Since the delay to calculate addresses and issue the subsequent loads can not be tolerated in this microbenchmark, false sharing misses have a major effect on performance. `CD` can calculate the next address by using data in local invalid blocks to issue the subsequent memory accesses early, overlapping false sharing misses. Even at 200 cycles, the speedup of `CD` is more than 45%. As communication latencies increase, the performance of `CD` degrades much more slowly than the baseline. At a 1000-cycle communication latency, the baseline system is about three times slower than `CD`, although the slopes of performance degradation become similar once the finite window and MSHR sizes prevent tolerance of longer latencies.

5.5.2 Coherence Decoupling Accuracy

In this section, we present speculation accuracies for a number of the coherence decoupling policies that are described in Section 5.3. Figure 5.5 shows the ratio of

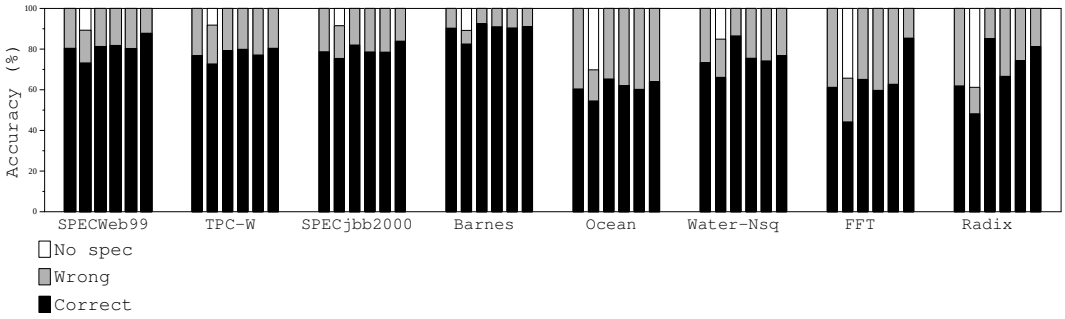


Figure 5.5: Accuracy of coherence decoupling (from left to right: CD, CD-F, CD-IA, CD-C, CD-N, CD-W)

correct to incorrect CD speculations (for all coherence misses) using a 4MB cache with 128-byte blocks, for a subset of the policies described in Section 5.3. In the `CD-N` experiment, we updated the invalid sharers after the first 5 writes to a line.

The `CD-F` policy is the only one to not speculate on all coherence misses, due to its filter which blocks low-confidence speculations. The base `CD` protocol makes more correct speculations than `CD-F`, but at the expense of more mispredictions. However, this simple protocol provides accuracies that approach those of many of the update protocols, due to silent stores and false sharing. For three commercial benchmarks and Barnes, the base `CD` protocol can predict correct values for more than 70% of coherence misses. Some of the update protocols lose accuracy by sending the update too early, changing an invalid line to a new value, after which the writer changes the value *back* (a temporally silent store) but may not broadcast the change, resulting in a mis-speculation.

Update-component protocols have better accuracy than `CD` for some benchmarks, but the improvement is modest. `CD-W` improves the prediction accuracy for FFT and Radix, but it does not increase the accuracy for the other benchmarks.

The CD-IA policy sees better accuracy for Water-nsq, FFT, and Radix, than CD-C, because the latter policy can not deliver a truly-shared value, if the value is not one of the values that can be encoded and sent along with the invalidation (-1, 0, or 1). Overall, coherence decoupling appears to have much better accuracies for the commercial workloads, with the simplest CD protocol performing as well as the more complex protocols, except on a few of the simpler scientific codes.

5.5.3 Coherence Decoupling Timing Results

We now consider timing simulation results for a system with coherence decoupling. Table 5.3 shows the speedups over the baseline system (which is the simple invalidation protocol with no coherence decoupling or speculation) for the range of policies described in Section 5.3. We model a flushing mechanism to recover from mis-speculations. The mechanism flushes all instructions younger in program order when the violation is detected (a "rolling flush") rather than waiting until the violation reaches the head of the reorder buffer. The rolling flush mechanism reduces the cost of speculation recovery, and is implemented in modern server processors such as IBM's Power5 [33].

The right-most column of Table 5.3 places an upper bound on the performance of coherence decoupling in the simulated system. In this model, all cache accesses that would have been coherence load misses are treated as hits. For TPC-W, the best-case speedup is 17.8%, providing only moderate opportunities for coherence decoupling speedups. SPECWeb99 and Ocean show larger benefits (34.6% and 34.5%). The only benchmark with an ideal coherence decoupling speedup of under 15% is Barnes, which is a mere 1.4% due to its negligible L2 miss rates.

Benchmark	CD	CD-F	CD-IA	CD-C	CD-N5	CD-W	Optimal
SPECWeb99	13.8%	11.0%	13.2%	13.1%	14.9%	18.0%	34.6%
TPC-W	1.2%	2.6%	2.3%	1.7%	1.4%	2.4%	17.8%
SPECjbb2000	16.6%	15.8%	13.5%	13.0%	17.1%	16.5%	26.3%
Barnes	0.6%	0.4%	0.7%	0.7%	0.8%	0.6%	1.4%
Ocean	6.9%	4.7%	8.2%	7.4%	6.0%	7.5%	34.5%
Water-Nsq	2.1%	1.7%	2.8%	3.5%	0.7%	5.4%	17.4%
FFT	5.1%	4.2%	6.1%	7.2%	4.6%	10.8%	21.4%
Radix	6.8%	3.6%	7.6%	8.8%	6.3%	12.0%	42.4%
Mean	6.6%	5.5%	6.8%	6.9%	6.5%	9.1%	24.5%

Table 5.3: Speedups for Coherence Decoupling

The accuracies of coherence decoupling are high, partially or fully tolerating a third to a half of coherence misses. The speedups reflect those results for several benchmarks; in particular, SPECJbb reaches over half of its ideal performance improvement for most of the policies. Overall, with only simple mechanisms, the base CD policy achieves a mean speedup of 6.6%, which is over a quarter of the ideal speedup. In larger-scale systems (and particularly CC-NUMA systems), the speedups will likely be much higher. In those systems, remote coherence latencies–especially those that take multiple hops across the network–will have a more deleterious effect on performance.

The update-based SCL protocols consume extra network bandwidth to increase prediction accuracies. Table 5.4 presents the network bandwidth overhead for `CD-IA` and `CD-N5`. In all update protocol experiments, we only transmit the updated words (not entire cache lines) to reduce bandwidth consumed. `CD-IA` incurs only small traffic increases (under 4%). `CD-N5` incurs much larger increases in traffic, with a range of 6% to 30% except for Barnes, which increases traffic by 95%. Due to Barnes' low L2 miss rates, however, that outlier has little effect on performance.

Benchmarks	CD-IA	CD-N5
SPECWeb99	3.6%	7.9%
TPC-W	3.9%	18.5%
SPECjbb2000	2.5%	2.5%
Barnes	2.7%	95.3%
Ocean	3.4%	6.1%
Water-Nsq	2.0%	28.1%
FFT	2.8%	10.5%
Radix	3.2%	8.2%

Table 5.4: Data traffic increase (%) for CD-IA and CD-N5

5.6 Comparing Coherence Decoupling to Transactional Memory

Transactional memory provides a new programming model for shared memory parallel programmers [41, 89, 40, 4]. In traditional shared memory programming models, programmers must identify conflicting accesses to critical regions, and protect the critical regions by serializing the accesses through locks. Such lock-based synchronization causes two problems: low programmer productivity and system performance losses. Programmers must use very fine-grained locks to avoid lock contentions, which complicates parallel programming. If locks are not optimized for fine-grained accesses, processors can contend to acquire locks, even if the data accessed in critical regions are mutually exclusive.

Unlike traditional shared memory programming models, transactional memory eliminates the need for locks by guaranteeing the serialization of conflicting accesses without explicit locks. Hardware detects conflicting accesses to the same data (Read/Write or Write/Write), and rolls back transactions, allowing one transaction to progress. Since transactional memory does not rely on explicit lock vari-

	Coherence Decoupling	Transactional Memory
Lock accesses	May be speculated	No locks
False sharing	Can hide	Can not hide
True sharing	May hide with updates	Can not hide
Access conflicts	Can not occur (lock)	Transaction rollback

Table 5.5: Comparison between coherence decoupling and transactional memory

ables to avoid access conflicts, it can eliminate locks and false contentions on locks completely.

Coherence decoupling may also improve performance for lock accesses. A lock is usually updated as a temporal silent store, with a locking store and a subsequent unlocking store. If a lock contention does not occur, coherence decoupling allows processors to execute critical regions speculatively. If there is any lock contention, speculatively executed instructions will be squashed except one processor which acquired the lock. All stores are buffered in the store queue of processors till they become non-speculative. However, unlike transactional memory, coherence decoupling still relies on serialization through lock variables to ensure mutually exclusive accesses to critical regions. Coherence decoupling can not mitigate lock contentions due to unoptimized coarse-grained locks.

Although transactional memory can completely eliminate locks, transactional memory can not reduce the true communication of shared data. Since transactional memory also uses caches for fast local accesses and coherence protocols for communication among processors, both false sharing and true sharing misses occur in transactional memory. The performance losses due to false sharing misses may be more severe with transactional memory. The false sharing misses may cause un-

necessary rollbacks of transactions, if two conflicting transactions accesses the block simultaneously.

Coherence decoupling can be combined to transactional memory to hide the effect of false and true sharing misses, if transaction conflicts do not occur. Misspeculation with coherence decoupling does not necessarily start rolling back the current transaction, since the requested block might have been updated before the current transaction started.

Table 5.5 summarizes the comparison between coherence decoupling and transactional memory.

5.7 Summary

This chapter considered the use of speculation to tolerate the long latencies of interprocessor communication in shared memory multiprocessors. The proposed approach, called *coherence decoupling*, breaks up the cache coherence protocol, which is used to implement coherent inter-processor communication, into a speculative cache lookup (SCL) protocol that returns a speculative value, and a coherence correctness protocol that confirms the correctness of the speculation. An early return of a (speculative) value allows further useful computation to proceed in parallel with the coherence correctness protocol, thereby overlapping long coherence latencies with useful computation. Furthermore, decoupling the SCL protocol, which returns a value from the protocol that ensures the correctness of the value, allows each protocol to be optimized separately. The SCL protocol can be optimized for performance since it does not have to ensure correctness; the coherence protocol can be simple since its performance is not paramount.

We implemented a variety of options for the two components of an SCL protocol: the read component and the update component. The basic read component returns the value from a matching invalid cache line for which the access permissions are not correct. Another option we measured was the addition of a confidence filter to determine when coherence decoupling should be employed, to reduce the number of mis-speculations. For the update component, we considered several variations of a canonical write-update protocol. These variations trade off the accuracy of speculation of the SCL protocol with the additional bandwidth required.

Using the MP-sauce full-system simulator running a set of commercial workloads and scientific workloads, our experiments showed that coherence misses are a significant fraction of total L2 misses, ranging from 10% to 80%, and averaging around 40% for large caches. Coherence decoupling has the potential to hide the miss latency for about 40% to 90% of all coherence misses, mis-speculating roughly 20% of the time.

We also measured the performance benefits of coherence decoupling. Several of the benchmarks are sensitive to coherence misses, so lower coherence latencies can improve performance. On these workloads, coherence decoupling was able to achieve modest improvements. One of the benchmarks is affected little by coherence misses and, unsurprisingly, coherence decoupling did not help in this case. These results suggest that coherence decoupling is generally able to overcome the performance drawbacks of false sharing and, furthermore, allow lower effective latencies even when true sharing is present.

We expect techniques like coherence decoupling to grow in importance for future processors and systems for several reasons:

- First, multiprocessors and/or multithreaded processors will be soon be ubiquitous; almost every future processing chip will employ some form of multiprocessing or multithreading. Rather than burdening programmers with having to reason about the performance effects of data sharing, architects can develop alternative techniques to overcome these performance impediments without burdening the programmer. Coherence decoupling is a technique that overcomes one such performance impediment (false sharing), and mitigates true sharing in some cases.

- Second, with increasing cache sizes, coherence misses will account for a larger fraction of all cache misses. This trend, coupled with increasing communication latencies, will cause the performance loss due to coherence misses to become a larger fraction of the overall performance loss. The performance loss for coherence misses will be magnified even further as other sources of performance losses (e.g., locks) are attenuated (for example, with speculative synchronization).

- Third, as communication latencies grow, there will be temptation to make coherence protocols more complex to reduce average latency. We believe that coherence protocols should be kept simple, relying on microarchitectural techniques to reduce communication-induced performance losses. Again, coherence decoupling is such a technique: the SCL protocol can allow the latency of the coherence protocol to be overlapped with computation that is likely to be useful.

- Finally, much of the hardware support required to support coherence decou-

pling is very likely to exist for other reasons — e.g., to overcome the performance limitations of sequential consistency, or to implement other speculative execution techniques. This fact will permit coherence decoupling to be implemented with less additional hardware and complexity.

In this chapter, among the three aspects of MP cache performance, we addressed the communication latency issue. Instead of forcing system designers and application programmers to reduce protocol latencies and fine-tune applications, coherence decoupling transparently hides long communication latencies with speculation. In the next chapter, we address the last aspect of the MP cache performance: communication bandwidth.

Chapter 6

Subspace Snooping: Increasing Snoop Tag Bandwidth

In the previous two chapters, we proposed and evaluated two techniques: shared caches to reduce off-chip misses in CMPs, and coherence decoupling to hide communication latencies with speculation. In this chapter, we propose a new type of coherence protocol called *subspace snooping*, which increases the effective bandwidth of snooping coherence protocols.

The expense of supporting cache coherence is what has limited very large-scale shared-memory multiprocessors. The largest message-passing clusters now number in the tens of thousands of processors, but shared-memory machines have not kept pace. This divergence is in part due to the financial cost of developing custom hardware for large-scale shared-memory machines, and in part due to the complexity of the protocols.

The two broad classes of coherence protocols, snooping protocols and di-

rectory protocols, have traditionally targeted different scales of systems. Snooping systems offer low-cost, simple coherence at small system scales (two to a few tens of processors). Directory protocols have been built to scale much higher, but at great cost and complexity. Directory protocols also suffer from the latency of numerous point-to-point messages in a large-scale system, which can be reduced at the expense of additional protocol complexity.

Snooping protocols do not scale to large numbers of processors for three major reasons: bus bandwidth, bus speed, and snoop tag bandwidth. As more processors snoop a single address bus, the traffic on the bus grows linearly with the number of processors. Sun's Wildfire system mitigated this problem by providing multiple address-interleaved snoop buses [38] supported by a point-to-point data transfer network. This solution allowed the number of processors to increase, but the bus backplane must still be routed to all snooping processors, resulting in long traces and decreasing the bus speed. Finally, perhaps the worst factor for scaling is snooping tag bandwidth. As the number of processors grows, the number of snoops that happen system-wide grow as $O(n^2)$. For each request that each processor puts on the snooping bus, n processors must snoop the request. For multiple interleaved address buses, providing the L2 (or L3) tag bandwidth for snooping, even with replicated tag banks, quickly becomes a performance and energy bottleneck.

The ideal large-scale hybrid protocol would allow many processors to share a high-bandwidth, low-latency bus, but also to have scalable snoop tag bandwidth. Viewed another way, in the ideal system, processors would only snoop the bus transactions for operations on data that they were likely to be sharing.

We define *subspaces* as regions of data that are consistently shared by a

stable subset of processors. These subspaces can be dynamically evolving, so long as they are stable for sufficiently long to be useful. For example, the faces on a cubic data decomposition in a CFD (Computational Fluid Dynamics) code are shared by stable pairs of processors for the duration of the application, assuming that it is a non-adaptive code.

Subspace snooping is a new type of snooping protocol that attempts to exploit stable subspaces to improve snooping scalability and energy efficiency. In a subspace snooping implementation, the system would provide some number of data channels (address buses being one example) that could all be snooped. However, an individual processor would likely only have enough snoop tag bandwidth to snoop a subset of the channels. Ideally, stable sets of processors sharing a subspace would allocate that subspace to a single channel that all of the participating processors would share. Processors not sharing that subspace would be snooping other channels, not incurring the energy or delay costs of snooping data that they were guaranteed not to share.

Subspace snooping protocols are likely to be best in systems where the aggregate available bus bandwidth exceeds the snooping tag bandwidth, and in which large numbers of processors can be partitioned into regular and fairly stable sharing sets. Optical buses may be an excellent match for the former constraint. Many processors may share them, their latency scales significantly better with added processors than do electrical buses, many channels may be implemented using wave- or time-division multiplexing, and, most important, the available bandwidth on the optical link greatly exceeds what a set of snoop tags would be able to follow.

6.1 Scaling Snooping Cache Coherence

Traditionally, the bandwidth of address and data buses has limited the scalability of snooping cache coherence systems. Bus bandwidth has been increasing with faster system clocks, wider buses, split transactions, separate address/data buses and switched networks. However, the cost of wide electrical buses still limits the expansion of snooping coherence systems. Recently, there has been a significant improvement of the cost-performance of optical interconnects for multiprocessor systems. Optical interconnects can provide large bandwidth with relatively low cost and energy consumption, thus improving the scalability of snooping coherence systems. As optical interconnects can increase bus bandwidth significantly, snoop tag bandwidth will still limit the bandwidth of snooping coherence protocols, since all snoop tags should respond to each bus transaction.

In this section, we first describe optical interconnection technologies for multiprocessors. As bus bandwidth increases, the limited bandwidth and energy consumption of snoop tags will become a bottleneck. In the second part of this section, we discuss how snooping protocols will be limited by snoop tag bandwidth in terms of both energy and performance scalability.

6.1.1 Optical Interconnection Technologies for Snooping Cache Coherence

Recently, optical interconnection technologies for multiprocessors have improved significantly. Arrays of Vertical Cavity Surface Emitting Lasers (VCSELs) and arrays of photodectors (PDs) provide inexpensive and fast electro-optic and opto-electric conversion [73, 112]. In current technologies, a VCSEL can transmit 3-5 Gb/s and an

array of the VCSELs can achieve 200-300 Gb/s transmission rates [65]. In addition to traditional optical fibers, polymer waveguides enable dense board-level optical interconnections [112]. These optical device advances have made optical interconnects a high-bandwidth alternative for traditional electrical buses in multiprocessors.

The optical interconnects have two advantages over traditional electrical wires for snooping multiprocessor systems. First, the bandwidth/cost and bandwidth/power of optical interconnects are superior to those of electrical wires. In optical links, furthermore, multiple wavelengths can co-exist in a single optical fiber [78, 12]. Such wave-length division multiplexing (WDM) can multiply the interconnection bandwidth without adding more physical links. The number of wavelengths are typically limited by the cost and latency of electro-optic/opto-electric conversion devices. Optical links for wide area networks, in which the bandwidth per distance is more important than the conversion latency of optical and electrical signals, use a dense WDM with tens or even hundreds of wavelengths. However, in current technologies, the optical interconnects for multiprocessors are constrained by the conversion latency and can support coarse-grained WDMs with 4-12 wavelengths, but the number of wavelengths are likely to increase as the optical technologies mature.

Second, optical interconnection can broadcast signals efficiently with passive components. In optical interconnects, high fan-outs at high frequencies are feasible. A passive star coupler can provide all-to-all connectivity over hundreds of nodes. This broadcast capability makes the optical links desirable for the address networks of snooping cache coherence. In current technologies, hundreds of fan-outs are possible without a significant loss of signal strength. The optical broadcast can be

more power-efficient than hierarchical packet switched buses that are widely used for snooping address buses. Furthermore, the topology of optical interconnections can be simple and easily extended for more processors. Current electrical interconnections in multiprocessors use switched networks, the topology of which is fixed and not modularized. Optical interconnects can connect multiple processors with passive star couplers, which simply divide light signals into multiple links [35].

Recent studies have shown that off-the-shelf parallel optical fibers can be used for multiprocessor interconnections. The Lambda-connect project at Lawrence Livermore National Laboratory and Multi-wavelength Assemblies for Ubiquitous Interconnects (MAUI) demonstrated that multi-wavelength parallel optical interconnect (MPOI) can greatly improve the bandwidth of multiprocessor systems [85, 60]. In both projects, parallel multi-mode fiber ribbon cables with 10-12 wires have been used for short distance optical interconnects with 4 channel WDM. In MAUI, an array of 48 VCSELs and 48 photodetectors were used to construct 12 wires and 4 wavelengths/wire. For each fiber, signals from 4 VCSELs with different wavelengths are multiplexed. Four photodectors in the receiving port pick up four different wavelengths.

The large bandwidth and efficient broadcast capability of optical links have led to several research investigations for optical snooping buses [83, 10, 24]. In SPEED, optical buses have multiple WDM channels and the channels are divided into a shared channel and multiple private channels. Bus requests for writes (ownership request) are sent over a shared channel, and read requests are sent to memory through private channels [37]. SYMNET used passive Y-splitters/couplers to connect processing nodes in a tree [65]. SYMNET modified a coherence protocol to

eliminate combined snoop results from all processors. Future optical interconnects will provide effective broadcast networks with enormous bandwidth. In the next section, I will show how snoop tags can hinder the scalability of snooping cache coherence.

6.1.2 Power Limitation of Snoop Tag Lookups

As the number of processors in multiprocessors increases to solve larger problems, the size of data set also scales with the number of processors. Therefore, to support the increased coherence traffic, address bus bandwidth should increase linearly with the number of processors. However in snooping cache coherence, all cache tags must respond to bus requests, so the total number of snoop tag accesses will increase quadratically with the number of processors. We assume the data set size scales linearly with the number of processors (N_p), so L2 misses per processor, $Misses_{L2}$ is a constant, independent of N_p. With N_p and $Misses_{L2}$, the number of bus transactions is $N_p \times Misses_{L2}$. Since every tag should be snooped for each L2 miss, the number of total tag lookups is computed:

$$TotalTagLookups = Np \times Np \times Misses_{L2}$$

The quadratic increase of the number of tag snoops will consume a significant power to access tag arrays and drive signals through I/O pins. A recent study showed that a significant fraction of L2 cache energy is consumed for snoop tag lookups [75]. We used a similar analytic method to show how the energy consumption of snoop tags will increase as the number of processors in SMPs increases. In the analysis, each processor incurs the same number of L2 misses and the number of L2 misses per processor is constant across different numbers of processors, assuming that the data

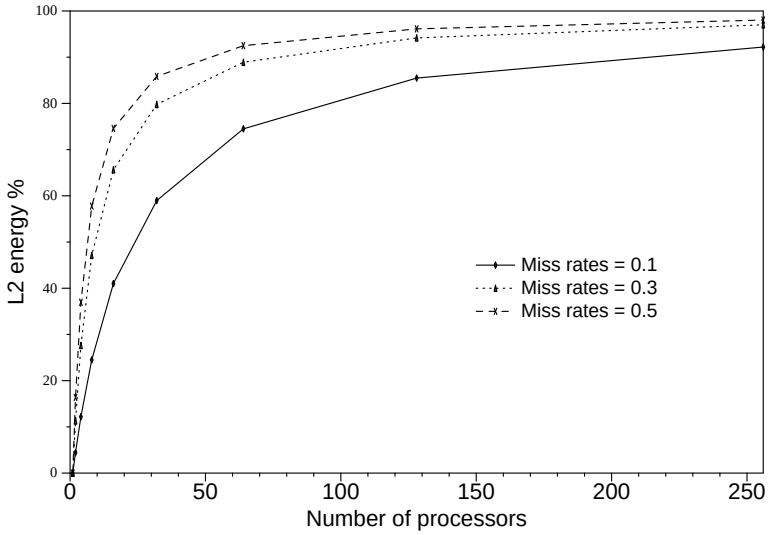


Figure 6.1: Energy consumed by snoop tag accesses (% of total dynamic cache energy)

set scales to the number of processors. $Energy_{Data}$ and $Energy_{Tag}$ are the energy used for each access to the data array and tag array in the local cache. $RemoteHits$ is the rate for missed blocks to be found in one of the remote caches. The model for the ratio of snoop energy to the total energy ($Ratio_{Snoop}$) is as follows:

$$Energy_{DataAll} = Energy_{Data} \times (1 + Misses_{L2} \times RemoteHits)$$

$$Energy_{Snoop} = Energy_{Tag} \times (Np - 1) \times Misses_{L2}$$

$$Energy_{TagAll} = Energy_{Snoop} + Energy_{Tag} \times (1 + Misses_{L2})$$

$$Ratio_{Snoop} = Energy_{Snoop}/(Energy_{Data} + Energy_{TagAll})$$

Figure 6.1 shows the fraction of L2 energy consumed for tag snooping, as the number of processors increase. At 128 processors, with 10% of L2 miss rate for each processor, tag lookups consume more than 80% of the total L2 dynamic energy. With the miss rates of 20 and 30%, snoop tag lookups consume more than 95% of

Bus Systems	SGI Challenge	Sun Gigaplane-XB	Sun Fireplane
Bus Bandwidth	47.6M/sec	167M/sec	150M/sec
Maximum Processors	36	64	24
Processor Clocks	150MHz	300MHz	750MHz
Transactions/1K cycle/processor	8.8 transactions	8.6 transactions	8.3 transactions

Table 6.1: Bandwidth requirements for past SMP systems

the L2 dynamic energy at 128 processors.

6.1.3 Performance Limitation of Snoop Tag Bandwidth

In conventional snooping protocols, snoop tag bandwidth is tightly coupled with bus bandwidth. Since the bus bandwidth has been smaller than the potential bandwidth of snoop tags, the bus bandwidth has been the focus of research to scale snooping coherence [11, 70, 66]. However, optical links with WDM can greatly improve the address network bandwidth, but the snoop tag bandwidth is constrained by processor clock speed and pin bandwidth. Although multi-ported snoop tags can increase the bandwidth, multi-porting is not a scalable solution due to large area and energy overheads. Furthermore, processor clock speed improvement has been slowed down recently due to power consumption even in high performance systems. For example, massively parallel systems, such as BlueGene/L, use a slow processor clock speed but scale out the system with many processors [29].

Table 6.1 shows a simple extrapolation of snoop bandwidth for three SMP systems. Estimated from the snoop bandwidth and processor clock speed data, the last row shows how much bandwidth the system designers assigned to each processor. We used processor clock speeds at the years when the buses were first introduced. For all three systems, approximately 9 bus transactions/1K cycles should be sus-

tained for each processor. If we extrapolate this requirement to larger systems, about 110 processors will need a coherence bandwidth of 1000 transactions/1K cycle, which is the maximum snooping speed a perfectly pipelined single-ported tag can provide. Even when optical buses can support scaled bandwidth for a large number of processors, snoop tag bandwidth, essentially limited by processor speed, limits the size of the system to 110 processors.

As discussed in this section, optical technologies can provide broadcast networks with enormous bandwidth. However, the energy consumption and limited bandwidth of snoop tag accesses will limit the scalability of SMPs.

6.1.4 Related Work

There have been several studies for reducing snoop-induced energy consumption. In *Jetty*, coarse-grained filters between snoop tags and a bus are used to discard snoops on the blocks which are not in caches [75]. The coarse-grained filters are much smaller than the cache tags, consuming less energy. *RegionScout* exploits the observation that there are large continuous private regions, which are not shared by other processors [74]. The RegionScout filters detect private regions and use the detected regions to reduce unnecessary snoop tag lookups.

Previous work on coherence bandwidth problem in multiprocessors includes using both snooping coherence and directory protocols adaptively in one system [70], multicasting snoop requests based on the prediction of potential sharers [11, 66], and using a token-based coherence mechanism on fast but un-ordered switched networks [68].

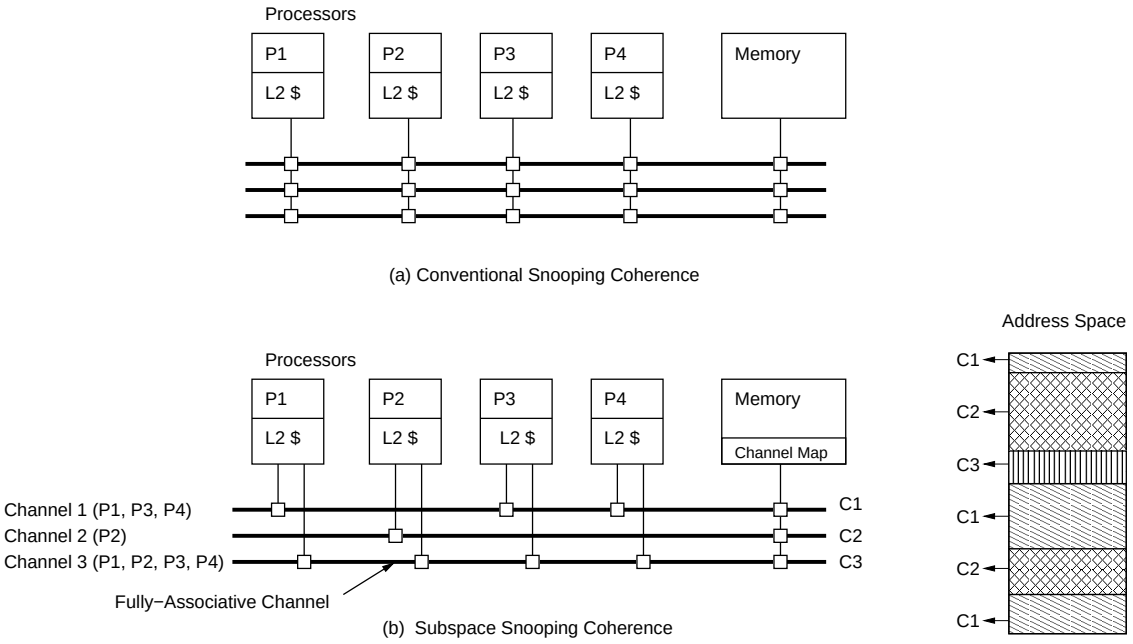


Figure 6.2: Conventional snooping and subspace snooping coherence

6.2 Subspace Snooping Coherence Architecture

Subspaces are regions of data that are consistently shared by a stable subset of processors. A subspace is represented by a partition of address space (data) and a subset of processors (sharers), which share the address partition. To maintain coherence for a cache block, bus requests should be delivered to the processors in the subspace the block is mapped to. Subspaces are not static, but may dynamically evolve during program execution.

Figure 6.2 describes conventional snooping and subspace snooping coherence. In conventional snooping protocols, all processors snoop every bus request, incurring a large number of snoop tag lookups. Even if the address bus consists of multiple

address-interleaved buses, processors must snoop all the buses.

In subspace snooping, we divide the available bus bandwidth into multiple channels. Coherence messages for a subspace are delivered through the channel designated to the subspace. The processors in a subspace snoop the channel dedicated to the subspace. In Figure 6.2 (b), there are three channels and each processor snoops only two channels. For example, P1 snoops only channel 1 and 3. The channel 1 is snooped by P1, P3, and P4, so a bus transaction on the channel 1 will cause three tag lookups.

The last channel is reserved for a fully associative channel, which is snooped by all processors. The fully associative channel covers the subspace shared by all processors or the subspaces which can not be mapped to any other channels.

There are three key issues to design subspace snooping coherence:

- *Mapping data and processors to subspaces:* The physical address space is partitioned and mapped to subspaces. Subspace snooping should have a mechanism to maintain the address mapping and to route bus requests to correct channels.

- *Guaranteeing correctness:* Subspace snooping should guarantee the correctness of coherence by preventing processors from caching addresses which the processors do not snoop. Since subspaces are not static, processors may attempt to access addresses mapped on un-snooped channels.

- *Forming optimized subspaces:* Subspaces should be formed to minimize snoop tag lookups. Subspace snooping should identify frequently communicating processors and group them to share a channel.

6.2.1 presents a mapping mechanism to ensure that requests are sent through

correct channels. In 6.2.2, I will discuss how to maintain correctness when subspaces evolve during execution. 6.2.3, compares subspace snooping to directory protocols.

6.2.1 Logical Channels and Channel directory

In optical interconnects, channels can be created in various ways. First, when an optical bus consists of multiple optical fibers, each channel can use a separate set of fibers. Second, for the same set of fibers, wavelength division multiplexing (WDM) can make separate channels, assigning a wavelength to each channel. Third, for the same set of fibers and a wavelength, time division multiplexing (TDM) can use different time slots to distinguish different channels. Any combination of these techniques can be used to create logical channels. With these techniques, snoop interfaces can discard requests on un-snooped channels without consuming any pin bandwidth.

Subspace snooping partitions the physical address space and maps the subspaces to logical channels. A *channel directory* contains the mapping between addresses and channel ids, similar to the directories in directory protocols. For each block of memory, the channel directory should contain the corresponding channel id. The mapping can be stored in a separate memory or ECC bits of external DRAM memory [30]. The bit width of a channel directory entry is the logarithm of the number of channels, which is smaller than that of directory protocols. Subspace partitions can change as subspace snooping adapts to actual memory sharing patterns. Whenever an address is moved to another channel, the channel directory should be updated synchronously. However, unlike directory protocols that must constantly keep track of accurate sharers, the subspace changes are much less

frequent than the directory updates.

For a bus request, the channel directory should route the request to a correct channel. In our baseline protocol, a node must first send a bus request to the channel directory, which will put the request on the correct channel. Due to the two-step bus transactions to route bus requests on correct channels, the baseline subspace snooping requires three-hop cache-to-cache transfers, increasing communication latencies. The initial bus requests from processors to the channel directory do not need to use the broadcasting address bus. Instead, subspace snooping uses un-ordered data networks to send the initial requests to the channel directory, to avoid using the broadcast bandwidth of address bus.

Subspace snooping reduces the latency increase from indirect bus accesses by embedding channel ids in cache tags. Cache tags store the current channel ids of cached blocks. For upgrade transactions, which change a block state from a shared to a modified state, issuing processors can put bus requests on correct channels directly by using the channel ids. However, for read requests for sharing or ownership, issuing processors can not know correct channels, since the missed blocks are not in the local cache. For such misses, issuing processors conservatively send requests to the channel directory, which will queue the requests on correct channels.

An extended protocol I developed can predict a channel and send a request directly on the predicted channel. The channel directory checks the speculative bus request. If a bus request is on a wrong channel, the channel directory will re-route the request to the correct channel. If prediction is incorrect, this speculative technique can waste snoop tag bandwidth since a request can cause snooping on two channels. We use a simple prediction mechanism using the information in invalid

cache blocks. As shown in Chapter 5, some invalidated blocks remain invalid in local caches. The performance protocol uses the channel id in the invalid blocks. If addresses match in the cache tags but the block state is invalid, the performance protocol uses the embedded channel id to predict a channel.

6.2.2 Guaranteeing the Correctness of Subspace Snooping

To guarantee the correctness of subspace snooping, one condition must be satisfied: a node must snoop a channel, if it caches any address mapped to the channel. If a processor attempts to read a block into its cache and does not snoop the channel mapped to the block, the correctness condition can be violated. To avoid violations, subspace snooping can take one of two actions: 1) move the conflicting address to another channel, which the requesting processor already snoops, or 2) make the processor snoop the new channel. In both cases, there is some cost to resolve such channel conflicts.

To move the mapping of an address to a new channel, the address should be invalidated from all the caches that snoop the old channel, if the caches do not snoop the new channel too. The block can not reside in the caches, if the caches no longer snoop the channel to which the block is newly mapped. If a request is an upgrade or read for ownership, such invalidation does not cause any performance degradation, since the request will invalidate the block anyway. However, if a request is a read for sharing, the invalidation may cause subsequent misses from other caches.

If a block is shared by the processors that do not have a common channel, the block will keep causing mapping conflicts and block invalidations. Frequently conflicting addresses are moved to the fully-associative channel (FA channel). The

FA channel is a safety channel to map the conflicting blocks, which otherwise can not be mapped to other channels. However, the fully-associative channel consumes the same snoop tag bandwidth as conventional snooping protocols.

Subspace snooping restricts the number of channels snooped by each processor. If a processor needs to snoop a new channel, it should stop snooping one of the current channels. Disconnecting a processor from a channel is a costly operation. The processor should flush any block in its local cache, if the block is mapped to the channel to be disconnected. The channel ids in the cache tags are checked and blocks are flushed if the blocks are mapped to the disconnected channel.

6.2.3 Comparing Subspace Snooping to Directory Protocols

Directory protocols can provide higher scalability than traditional snooping protocols. However, due to the complexity of protocols and the overheads for the directory, the directory protocols have not replaced snooping protocols. Subspace snooping mitigates the snoop tag limitation of snooping protocols, with the simplicity of snooping protocols. In this section, we compare subspace snooping to directory protocols:

- Subspace snooping uses broadcast buses and caches are snooped atomically through the buses. Subspace snooping may avoid the protocol complexity of directory protocols.

- Each cache maintains coherence states in tags. Unlike the directory in directory protocols, the channel directory does not have sharing states, and thus the size of channel directory is much smaller than that of the directory in directory protocols.

- Increasing the granularity of coherence in directory protocols can degrade the performance significantly due to false sharing. Subspace snooping decouples sharing states from channel mapping states. Subspace snooping may increase the channel mapping granularity without incurring false sharing.
- Since subspace mapping changes less frequently than sharing states in directory protocols, predicting the channel id may be more accurate than predicting sharers for directory protocols. Such channel prediction allows two-hop cache-to-cache transfers.

6.3 Subspace Snooping Protocols

In this section, we present two subspace snooping protocols. Our baseline protocol always sends requests, except upgrade requests, to the channel directory first. After receiving requests, the channel directory forwards them to correct channels. Our performance protocol predicts channels for missed blocks and broadcasts requests through the predicted channels. The channel directory will correct the speculation of channel ids. In 6.3.3, we present our policy to form subspaces to reduce conflicts as well as snoop tag lookups.

6.3.1 Conflict Resolution

A mapping conflict occurs if a processor attempts to read a cache block, and the block is mapped to a channel that the processor is not snooping. In subspace snooping, the maximum number of channels a processor can snoop (snoop count) is limited, and in stable states, processors do not have free snoop counts. Therefore, if a processor needs to snoop a new channel, the processor should stop snooping one

of the current channels. However, as discussed in 6.2.2, detaching a processor from a channel is a very expensive operation requiring cache flushing.

For a mapping conflict, we change the mapping of the conflicting block to one of the channels the requesting processor is snooping. The channel directory informs the change (new channel id) to the processors on the old channel. The processors on the old channel should invalidate the block, if they are not snooping the new channel.

If a block causes too many conflicts, we map the block to the fully associative channel. In the channel directory, we add a 2-bit counter for each block, recording the history of conflicts. Note that moving the mapping of an address to the fully-associative channel, does not cause any block invalidation, since all processors snoop the FA channel. However, if too many blocks are mapped to the FA channel, conflicts may decrease, but snoop tag lookups will increase.

The mapping conflicts never occur for upgrade requests. For a upgrade transaction, the requesting processor already has the shared copy, so the processor must be snooping the current channel of the block the processor attempts to upgrade.

6.3.2 Baseline Subspace Snooping Protocol

To eliminate unnecessary accesses to the channel directory, the current channel ids of blocks are appended to cache tags. For upgrade transactions (changing a block state from shared to modified), requesting processors can put upgrade requests on correct channels by using the embedded channel ids.

For read misses, requesting processors do not know the correct channels for the missed blocks. Therefore, bus requests for reads are always forwarded first to

the channel directory. For such forwarding, our implementation does not use the address broadcasting bus. Instead, the bus request is sent through a point-to-point data network to the channel directory. The forwarding does not need to use the address bus, since request serialization occurs when the channel directory puts an actual bus transaction on the bus.

As discussed in 6.3.1, if a requesting processor is not snooping on the channel the requested block is mapped to, the channel directory starts the conflict resolution procedure. The channel directory adds the new channel id to messages when it forwards the request to the old channel.

6.3.3 Performance Subspace Snooping Protocol

The performance protocol improves the baseline protocol by broadcasting requests on predicted channels. The channel directory, which snoops all channels, verifies the predicted channels. If the predicted channel is incorrect, the channel directory initiates the second broadcast on correct channels. If the prediction is correct, the performance protocol reduces the latencies from the baseline protocol. If the prediction is incorrect, unlike the baseline protocol, the performance protocol may cause extra snoop tag lookups at the initially failed broadcasts of requests.

To predict the channel ids for read misses, we use a simple mechanism using invalid blocks. As we have shown in Chapter 5, many invalidated blocks remain invalid till processors access them again. If a missed block is in the invalid state, we use the channel id in the local cache, and place the request on the predicted channel. However, if the requesting processor does not snoop the predicted channel currently, the request is sent to the channel directory conservatively.

6.3.4 Policy for Forming Subspaces

As discussed in section 6.2.2, detaching a processor from a channel is a costly operation. The goals of dynamic channel mapping are: 1) to minimize snoop tag lookups by mapping the smallest set of processors on each channel and 2) to minimize conflicts (invalidations) and cache flushings. Our mapping policy uses a simple greedy algorithm to find a good processor mapping.

The mapping policy uses an NxN frequency matrix to keep track of communication frequencies between a pair of processors. The communication statistics are collected from snoop responses for each bus request. Using the communication frequencies, processors are grouped and mapped to channels.

Based on the past communication patterns, processor mappings can be adjusted periodically for reorganization. To minimize unnecessary flushings, the mapping policy chooses the processors which are not communicating with other processors in the same channel, and remove the processors from the channel. The processors are moved to other channels, if the processors communicate more with processors on the new channels.

6.4 Experimental Results

6.4.1 Methodology

We ran our experiments with the Augmint simulator [82]. The Augmint simulator instruments x86 binaries and generates memory references. A back-end simulator simulates the memory systems with multiple logical channels. The Augmint simulator does not simulate instruction caches, but the effect of instruction caches is

Application	Dataset/parameters	Memory usage	L2 miss rates	No. of bus accesses
AppBT	36x36x36 grid	32MB	0.11	2.5M
Barnes	128K particles	20MB	0.01	2.5M
FFT	1M data points	49MB	0.44	1.8M
FMM	128K particles	73MB	0.13	6.2M
Ocean	258x258 grid	15MB	0.13	2.7M
Radix	2M numbers	16MB	0.23	1.3M
Water	4096 molecules	25MB	0.04	3.7M

Table 6.2: Application parameters for workloads

small in these scientific benchmarks, since all instructions fit in level-one instruction caches. The simulator has a simple in-order processor model which always executes one instruction every cycle, if all loads and stores are in the L1 data caches. Each L1 and L2 cache can have a maximum of one pending request at a time. The number of processors is fixed to 64. The L1 and L2 caches are 2-way 16KB and 8-way 512KB respectively. The block sizes are 64B for the L1 and L2 caches. We simulate seven scientific benchmarks, six from SPLASH-2 benchmark suite [107] and *AppBT*. *AppBT* is a shared memory version of BT in the NAS parallel benchmark suite [14]. Table 6.2 shows the data sets and application statistics.

We simulate subspace snooping protocols with 9, 17 and 33 logical channels. The last logical channel is always the fully-associative channel (FA channel). Each processor can snoop at most three channels for 9, four channels for 17, and five channels for 33 channels. For the results in this section, subspaces are formed with pair-wise communication frequencies between two processors. They are created after the initialization period. Since the execution times for the benchmarks are relatively short compared to real systems, we do not reorganize the channel mapping.

Applications	9 channels	17 channels	33 channels
AppBT	54%	63%	69%
Barnes	27%	26%	26%
FFT	74%	84%	90%
FMM	39%	47%	50%
Ocean	59%	71%	78%
Radix	61%	66%	66%
Water	23%	35%	37%

Table 6.3: Snoop tag lookup reduction (%) : 9, 17, and 33 channels

Application	AppBT	Barnes	FFT	FMM	Ocean	Radix	Water
Snoop Reduction (%)	63%	26 %	84%	47%	71%	66%	35%
Bus access increase (%)	13.8%	4.9%	3.7%	3.5%	4.7%	3.2%	4.9%
FA channel usage (%)	22.9%	64.4%	0.0%	40%	14.9%	24.8%	70.6%

Table 6.4: Performance characteristics of subspace snooping: 17 channels

6.4.2 Reducing Snoop Tag Lookups

Table 6.3 shows the reduction of snoop tag lookups with 9, 17, and 33 logical channels. All seven benchmarks show the snoop reduction of 23-90% across different numbers of channels. For Barnes and Water, the amounts of tag lookup reduction are small. For the two applications, our channel mapping algorithm could not form stable subspaces effectively for the majority of data, due to relatively irregular sharing patterns.

As the number of logical channels increases, the reductions increase significantly for the five applications except Barnes and Radix. As more logical channels are available, the mapping algorithm can find subspaces with smaller numbers of processors. For the five applications, the reductions increase by 10-20% from 9 channels to 33 channels. As the bandwidth and the number of wavelengths in optical

interconnects increase, subspace snooping will allow more fine-grained partitioning of processors, reducing more snoop accesses.

The dynamic channel mapping in subspace snooping can incur cache misses by invalidating cache blocks to maintain correctness for mapping conflicts. Such extra misses caused by invalidations will increase, if subspace snooping can not form a good set of subspaces. Table 6.4 shows the increase of total bus accesses due to mapping conflicts and the percentage of bus accesses through the fully associative channel.

Except AppBT, six other benchmarks show the low increases of bus accesses, less than 5%. For those applications, the effect of extra bus accesses are small, but AppBT has a relatively large 14% increase of bus accesses. When the mapping algorithm can not find stable subspaces, there are many mapping conflicts and the majority of communication blocks are mapped to the fully-associative channel. Barnes and Water, which have the lowest reduction rates, show very high fully-associative channel usages of 64-70%. As more bus accesses are mapped to the fully-associative channel, subspace snooping may reduce tag lookups less effectively.

6.4.3 Accuracy of Performance Subspace Snooping Protocols

The performance subspace snooping protocol enhances the baseline protocol by sending bus requests directly through predicted channels. However, incorrect prediction can waste snoop tag bandwidth, since a request is broadcast twice on two channels. Figure 6.3 shows the prediction accuracy for bus accesses with 17 channels. We break down bus accesses to four classes: upgrade, no prediction, correct prediction, and incorrect prediction. For upgrades, requests are always placed on correct chan-

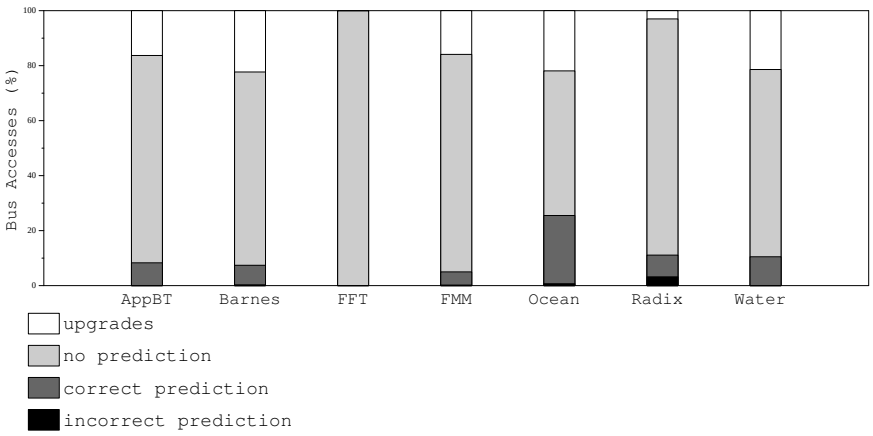


Figure 6.3: Channel prediction accuracy (17 channels)

nels directly without checking the channel directory. Therefore, upgrades do not use prediction. For read misses, the performance subspace snooping uses channel prediction only when there are invalid cache blocks for missed addresses and processors are snooping predicted channels. Otherwise, requesting processors send requests to the channel directory conservatively (no prediction).

For our benchmark applications, the ratios of upgrades are 20-30% except FFT and Radix. For 50-80% of bus accesses, the performance subspace snooping does not predict channels. The ratios of incorrect prediction are negligible except Radix with less than 5% of the total bus accesses. For five benchmarks (AppBT, Barnes, Ocean, Radix and Water), 10-25% of bus accesses can be predicted correctly by just using embedded channel ids in invalid cache blocks. Ocean shows the best ratio of 25%.

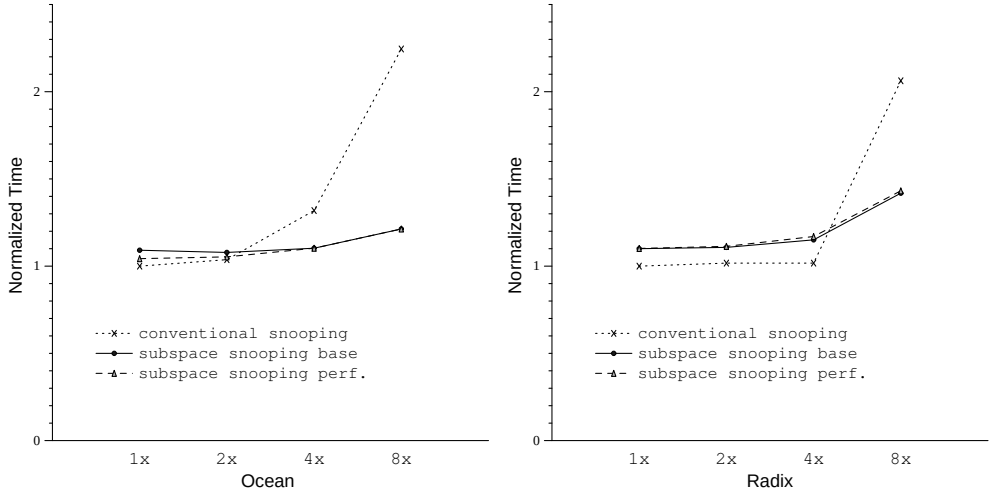


Figure 6.4: Performance scalability of subspace snooping

6.4.4 Performance Scalability

In this section, we present how performance can scale with more processors. Our simulation infrastructure does not allow scaling systems to hundreds of processors, due to simulation time, memory constraints and application scalability. To investigate the scalability of coherence protocols, we use an approximate method. Instead of scaling the number of processors, we decreased the available snooping tag bandwidth with a fixed number of processors (64), assuming the number of bus accesses will linearly scale with the number of processors and the consumption of tag bandwidth will increase linearly.

Figure 6.4 presents the results from selected two applications. The x-axis represents system scaling factors of 1, 2, 4, and 8 from a 64-processor system. We decrease the tag bandwidth accordingly by factors of 1, 2, 4, and 8 to approximate the system scaling. Among the seven applications, three applications (Ocean,

Applications	64B	128B	256B	512B	1KB
AppBT	63%	62%	60%	59%	55%
Barnes	26%	14%	4%	3%	3%
FFT	84%	86%	84%	84%	83%
FMM	47%	32%	24%	21%	18%
Ocean	71%	72%	67%	52%	32%
Radix	66%	66%	66%	59%	35%
Water	25%	25%	24%	24%	23%

Table 6.5: Snoop tag lookup reduction with varying block sizes (%) : 64B, 128B, 256B, 512B, and 1K granularity

Radix, and FFT) showed significant increases of the execution times at 4x and 8x scalings. The other applications do not show any performance degradation up to an 8x scaling factor, since the coherence bandwidth is sufficiently large for the applications. Figure 6.4 shows the results of Ocean and Radix from the three applications. For Ocean, when the systems are at 1x and 2x scaling factors, subspace snooping has lower performance than a conventional snooping protocol since the subspace snooping has the overheads of increased misses from mapping conflicts and increased latencies from indirect bus accesses through the channel directory. However, as less tag bandwidth is available, the burst traffic in Ocean degrades the performance of the conventional snooping protocol significantly. The results show the subspace snooping scales much better than the conventional snooping protocol beyond 4x and 8x scaling factors.

6.4.5 Address Mapping Granularity

One important difference of subspace snooping from directory protocols is to decouple coherence granularity from mapping granularity. In directory protocols, increasing block sizes will increase false sharing. Although there have been several studies

to mitigate the false sharing effect, false sharing has been an important performance issue in directory protocols [44]. Subspace snooping can still support small block sizes for coherence purpose, but the mapping granularity can be increased independently from the coherence block sizes.

Increasing mapping block sizes can be beneficial for two purposes: 1) it can reduce the channel directory size. A small channel directory will be faster, and thus reduce coherence latencies. 2) If channel ids are predicted for performance with external tables, a prediction table can cover more addresses with larger block sizes.

Table 6.5 shows how mapping granularity affects snoop reduction. As the granularity increases, mapping conflicts occur more frequently, and thus more addresses are mapped to the fully-associative channel. For all benchmarks except Barnes, increasing the granularity to 256B does not decrease the reduction of snoop tag lookups significantly. For AppBT and FFT, even 1KB mapping size has the tag lookup reductions comparable to 64B mapping sizes.

6.5 Summary

In this chapter, we have explored a new coherence protocol to expand the scalability of snooping coherence protocols. As bus bandwidth has been increasing by orders of magnitude, snooping protocols will soon face the limitation of snoop tag lookups. In traditional snooping protocols, every snoop tag must be looked up for each bus transaction. The energy consumption and bandwidth limitation of the snoop tag lookups will limit the scalability of snooping protocols. As optical interconnects can provide enormous bandwidth for broadcasting addresses, snooping protocols will not scale to hundreds of processors due to the limitation of snoop tag bandwidth.

Subspace snooping increases the snoop tag bandwidth by forming subspaces. Subspaces are regions of data stably shared by a subset of processors. Subspace snooping allows a bus request to be snooped only by the processors sharing the subspace. A hardware mechanism should recognize common communication patterns and form subspaces. We used pair-wise communication statistics between two processors to find subspaces.

We evaluated subspace snooping with the Augmint simulator. The results showed a 23-90% reduction of snoop tag lookups by using 9, 17, and 33 channels. Such snoop reduction allowed the systems to scale by factors of 4 and 8 for the three applications which suffered from the limited snoop tag bandwidth with a conventional snooping protocol.

While we believe that subspace snooping has potential for large-scale systems running regular, scientific applications, we do not believe that it can be made practical on electrical buses. With optics, however, the tradeoff space is quite different, and new types of coherence protocols may well arise if optical interconnects, particularly snooping ones, become widespread. We believe that many applications have stable sharing patterns that can be exploited transparently and much more efficiently, but it may require significantly larger systems (a higher number of processors), with more logical channels and bigger datasets than we can safely simulate. Nevertheless, we have shown that significant reductions in energy and tag contention are possible using subspace snooping.

Chapter 7

Conclusions

In this dissertation, we explored techniques to reduce the costs of communication in multiprocessors. The techniques solved three different aspects of communication problems in multiprocessors: cache misses in local caches, long coherence latencies, and communication bandwidth limitations. We evaluated the solutions in the context of chip-multiprocessor technologies, speculative out-of-order processors, and optical interconnection technologies.

7.1 Summary

We compared the area and performance trade-offs for CMP implementations to determine how many processing cores future server CMPs should have, whether the cores should have in-order or out-of-order issue, and how big the per-processor on-chip caches should be. We found that, contrary to some conventional wisdom, out-of-order processing cores will maximize job throughput on future CMPs. As technology shrinks, limited off-chip bandwidth will begin to curtail the number

of cores that can be effective on a single die. Current projections show that the transistor/signal pin ratio will increase by a factor of 6 between year 2005 and 2015. That disparity will force increases in per-processor cache capacities as technology shrinks, reducing the number of cores that would otherwise be possible. These conclusions emphasize the importance of reducing off-chip misses in future CMPs.

To reduce off-chip memory accesses, we evaluated shared cache designs for future CMPs. We proposed an organization for the on-chip memory system of a chip multiprocessor, in which 16 processors share a 16MB pool of 256 L2 cache banks. The L2 cache is organized as a non-uniform cache architecture (NUCA) array with a switched network embedded in it for high performance. We show that this organization can support the spectrum of degrees of sharing: *unshared*, in which each processor has a private portion of the cache, thus reducing hit latency, *completely shared*, in which every processor shares the entire cache, thus minimizing misses, and every point in between. We find the optimal degree of sharing for a number of cache bank mapping policies, and also evaluate a per-application cache partitioning strategy. We conclude that a static NUCA organization with sharing degrees of two or four work best across a suite of commercial and scientific parallel workloads. We also demonstrated that migratory, dynamic NUCA approaches improve performance significantly for a subset of the workloads at the cost of increased power consumption and complexity, especially as per-application cache partitioning strategies are applied.

As a novel latency hiding method for communication misses, we proposed a new technique called coherence decoupling, which breaks a traditional cache coherence protocol into two protocols: a Speculative Cache Lookup (SCL) protocol and

a safe, backing coherence protocol. The SCL protocol produces a speculative load value, typically from an invalid cache line, permitting the processor to compute with incoherent data. In parallel, the coherence protocol obtains the necessary coherence permissions and the correct value. Eventually, the speculative use of the incoherent data can be verified against the coherent data. Thus, coherence decoupling can greatly reduce — if not eliminate — the effects of false sharing. Furthermore, coherence decoupling can also reduce latencies incurred by true sharing. SCL protocols reduce those latencies by speculatively writing updates into invalid lines, thereby increasing the accuracy of speculation, without complicating the simple, underlying coherence protocol that guarantees correctness.

The performance benefits of coherence decoupling are evaluated using the MP-sauce simulator and a mix of commercial and scientific benchmarks. Our results show that 40% to 90% of all coherence misses can be speculated correctly, and therefore their latencies partially or fully hidden. This capability results in performance improvements ranging from 3% to over 16%, in most cases where the latencies of coherence misses have an effect on performance.

Snooping tag bandwidth is one of the resources that limits the number of processors that can participate in a cache-coherent snooping system. We evaluated a type of coherence protocol called subspace snooping, which decouples the snoop tag bandwidth from the address bus bandwidth. In subspace snooping, each processor snoops a set of logical channels, which are a subset of the total snoopable address buses in the system. Thus, each processor snoops a subset of the address space, reducing the number of tag matches required for a system of a given size. By dynamically assigning both processors and cache lines to channels, we support

dynamic formation of subspaces, with the goal of having only sets of processors that share data snooping on each given channel.

Subspace snooping aligns best with systems for which the address bus bandwidth greatly exceeds the snooping tag bandwidth. Snooping optical interconnects exhibit such characteristics, providing enormous transmission bandwidth, but which quickly become limited by snooping tag energy and bandwidth as the number of processors increases. Optical buses can be subdivided into logical channels using either wave-division or time-division multiplexing, making them good candidates for a subspace snooping implementation. We evaluated a range of subspace snooping protocols on a number of parallel scientific benchmarks, running on the Augmint simulator. The results showed 23-90% reductions of snoop tag lookups with 9, 17, and 33 channels. For three applications, which suffer from burst traffics with conventional protocols, subspace snooping allowed system scaling by factors of 4 and 8. Subspace snooping reduces the consumption of snoop tag bandwidth, but it does not increase the bus bandwidth. Therefore, it will increase system scalability only when optical interconnects can provide enough bus bandwidth.

7.2 Combining Three Techniques for Future Multiprocessors

Future large-scale shared memory multiprocessors will likely have many CMPs as building blocks, connected with high bandwidth inter-chip coherence protocols. The three techniques we proposed in this dissertation can be used together to enhance such future multiprocessors. Combining the techniques may generate synergistic

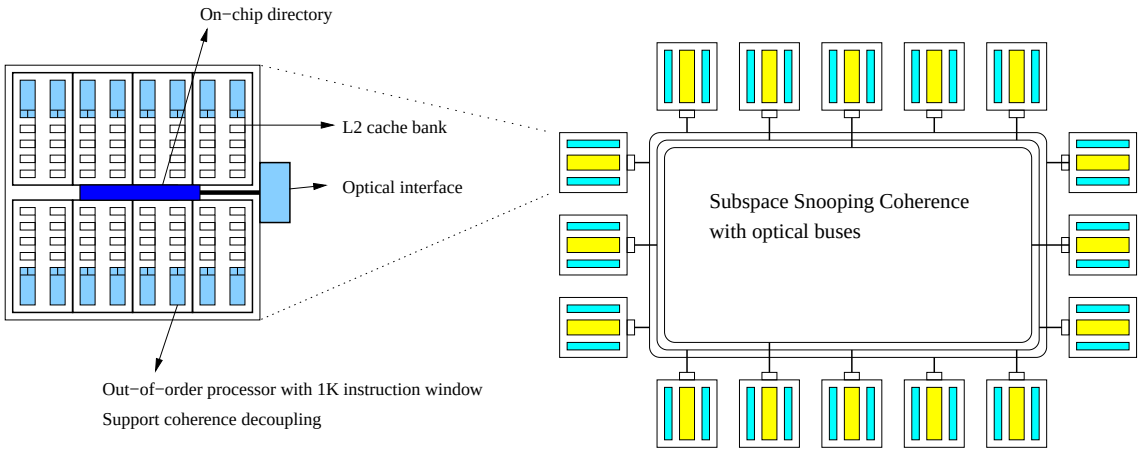


Figure 7.1: Future multiprocessor with combined three techniques

effect to improve the performance of multiprocessors.

Figure 7.1 shows a multiprocessor system projected to 32nm technologies with the three techniques. Each CMP consists of 16 processors with a large instruction window size of 1000 instructions. The 16MB on-chip cache consists of an S-NUCA array and an on-chip directory for coherence within the chip boundary. The S-NUCA cache can support per-line dual sharing degrees for private data and shared data.

Dual sharing degrees allow private data to be located close to processors, while replications of shared data are reduced. Processors can distinguish private and shared data either statically or dynamically. The static technique uses operating system support to identify shared pages at page-level. The operating system marks a private or shared flag for each page, and such sharing states of pages are also stored in translation look-aside buffers (TLBs). When a L1 miss occurs, a processor can identify the sharing state of the missed block (shared or private) from the flag

in the TLB entry.

The dynamic technique uses the on-chip directory to identify sharing flags. A block is considered private until more than one on-chip processors start sharing the block. The on-chip directory includes sharing flags. Once a block becomes a shared block, the sharing flag stays shared till the block is completely evicted from the on-chip cache. For a L1 cache miss, a processor can either send requests to the two banks (for private and shared degrees) serially for less power consumption or in parallel for better performance.

The CMPs are connected with high bandwidth optical buses. The optical interface of a CMP is connected to the directory. The on-chip directory responds to snoop requests as snoop tags for chip-to-chip coherence. The subspace snooping coherence system provides the chip-to-chip cache coherence with optical buses.

With a large instruction window size of 1000 instructions, speculative processors in CMPs can sustain long coherence latencies with coherence decoupling. Coherence decoupling supports both level-one cache misses and level-two cache misses. If a block is in invalid state in the L1 cache, the SCL protocol of coherence decoupling uses the invalid block in the L1 cache. The L1 coherence decoupling can hide communication latencies among processors on the same chip.

Combining the techniques can have both positive and negative effect on system performance. The effects of combining each pair of three techniques are as follows:

- CMP with shared NUCA and coherence decoupling: Coherence decoupling can hide on-chip coherence latencies by using invalid blocks in L1 caches. However, L1 cache capacity is much smaller than L2 cache capacity. Due to the small

capacity, in L1 caches, invalid blocks are more likely replaced before they are accessed again by processors, than in L2 caches, reducing the opportunities for L1-level coherence decoupling.

- Coherence decoupling and subspace snooping: The increased coherence bandwidth from subspace snooping can allow coherence decoupling to use update-based protocols aggressively. Sending speculative updates may become costly, if the coherence bandwidth is small and already saturated for the correctness protocol. The high bandwidth of optical buses and the increased tag bandwidth from subspace snooping can provide a large coherence bandwidth to support aggressive update-based SCL protocols. Subspace snooping allows a speculative update of a block to be sent only to the processors mapped to the subspace of the block, reducing snoop tag lookups for the processors receiving speculative updates to check snoop tags.

  Coherence decoupling can reduce the effect of long coherence latencies in large-scale multiprocessors, in which subspace snooping is effective. Although coherence latencies are long for such large-scale multiprocessors, a kilo-instruction window in future microprocessors will allows deep speculative execution to hide the long latencies.

- CMP with shared NUCA and subspace snooping: CMPs can generate more bus accesses than a single processor chip, and even if shared caches can reduce off-chip misses, CMPs will still issue more off-chip accesses as the number of processors on a chip increases. Subspace snooping can help scale multiprocessors with CMPs by increasing snoop tag bandwidth.

However, wrong scheduling of processes on CMPs can disrupt effective subspace formation. If processors in a CMP are mapped to different subspaces, the optical interface of the CMP must snoop many channels for the subspaces, reducing the effectiveness of subspace snooping.

Bibliography

[1] V. Agarwal, M. Hrishikesh, S. W. Keckler, and D. Burger. Clock rate versus IPC: The end of the road for conventional microarchitectures. In *The 27th Annual International Symposium on Computer Architecture*, pages 248–259, June 2000.

[2] V. Agarwal, S. W. Keckler, and D. Burger. The effect of technology scaling on microarchitecture structures. Technical Report TR-00-02, Department of Computer Sciences, University of Texas at Austin, May 2001.

[3] A. Alameldeen and D. Wood. Variability in architectural simulations of multi-threaded workloads. In *Proceedings of the 9th Int. Symp. on High-Performance Computer Architecture*, pages 7–18, Feb. 2003.

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

[5] C. Anderson and A. Karlin. Two adaptive hybrid cache coherency proto-

cols. In *Proceedings of the 2nd Int. Symp. on High-Performance Computer Architecture*, pages 303–313, Feb. 1996.

[6] J.-L. Baer and T.-F. Chen. Effective hardware-based data prefetching for high-performance processors. *IEEE Transactions on Computer*, 44(5):609–623, 1995.

[7] L. A. Barroso, K. Gharachorloo, and E. Bugnion. Memory system characterization of commercial workloads. In *The 25th Annual International Symposium on Computer Architecture*, pages 3–14, June 1998.

[8] L. A. Barroso, K. Gharachorloo, R. McNamara, A. Nowatzyk, S. Qadeer, B. Sano, S. Smith, R. Stets, and B. Verghese. Piranha: A scalable architecture based on single-chip multiprocessing. In *The 27th Annual International Symposium on Computer Architecture*, pages 282–293, June 2000.

[9] B. M. Beckmann and D. A. Wood. Managing wire delay in large chip-multiprocessor caches. In *37th International Symposium on Microarchitecture (MICRO)*, December 2004.

[10] A. F. Benner, M. Ignatowski, J. A. Kash, D. M. Kuchta, and M. B. Ritter. Exploitation of optical interconnects in future server architectures. *IBM Journal of Research and Development*, (4/5), 2005.

[11] 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, pages 294–304, Washington, DC, USA, 1999. IEEE Computer Society.

[12] C. Brackett. Dense wavelength division multiplexing networks: Principles and applications. *IEEE Journal of Selected Areas in Communications*, (6), Aug. 1990.

[13] D. Burger and T. M. Austin. The SimpleScalar tool set version 2.0. Technical Report 1342, Computer Sciences Department, University of Wisconsin, June 1997.

[14] D. Burger and S. Mehta. Parallelizing appbt for shared-memory multiprocessors. Technical Report 1308, Computer Sciences Department, University of Wisconsin, September 1995.

[15] A. Charlesworth. Starfire: Extending the smp envelope.

[16] Z. Chishti, M. Powell, and T. N. Vijaykumar. Distance associativity for high-performance energy-efficient non-uniform cache architectures. In *The 36th Annual International Symposium on Microarchitecture (MICRO)*, pages 55–66, December 2003.

[17] Z. Chishti, M. D. Powell, and T. N. Vijaykumar. Optimizing replication, communication, and capacity allocation in cmps. In *Proceedings of the 32nd annual international symposium on Computer Architecture*, 2005.

[18] A. L. Cox and R. J. Fowler. Adaptive cache coherency for detecting migratory shared data. In *Proceedings of the 20th Int. Symp. on Computer Architecture*, pages 98–108, May 1993.

[19] R. Crisp. Direct Rambus technology: The new main memory standard. *IEEE Micro*, 17(6):18–27, December 1997.

[20] D. E. Culler and J. P. Singh. *Parallel Computer Architecture*. Morgan Kaufmann, 1999.

[21] F. Dahlgren. Boosting the performance of hybrid snooping cache protocols. In *Proceedings of the 22nd Int. Symp. on Computer Architecture*, pages 60–69, June 1995.

[22] F. Dahlgren, M. Dubois, and P. Stenström. Combined performance gains of simple cache protocol extensions. In *Proceedings of the 21st Int. Symp. on Computer Architecture*, pages 187–197, Apr. 1994.

[23] W. J. Dally, M.-J. E. Lee, F.-T. An, J. Poulton, and S. Tell. High performance electrical signaling. In *The Fifth International Conference on Massively Parallel Processing Using Optical Interconnections*, June 1998.

[24] P. W. Dowd, J. A. Perreault, J. Chu, J. C. Chu, D. C. Hoffmeister, and D. Crouse. LIGHTNING: a scalable dynamically reconfigurable hierarchical WDM network for high-performance clustering. In *Proc. of the Fourth IEEE Int'l Symp. on High Performance Distributed Computing (HPDC-4)*, pages 220–229, 1995.

[25] M. Dubois, J. Skeppstedt, L. Ricciulli, K. Ramamurthy, and P. Stenström. The detection and elimination of useless misses in multiprocessors. In *Proceedings of the 20th Int. Symp. on Computer Architecture*, pages 88–97, May 1993.

[26] B. Falsafi, A. R. Lebeck, S. K. Reinhardt, I. Schoinas, M. D. Hill, J. R. Larus,

A. Rogers, and D. A. Wood. Application-specific protocols for user-level shared memory. In *Supercomputing*, pages 380–389, Nov. 1994.

[27] R. Farjad-Rad, C.-K. K. Yang, and M. Horowitz. A 0.3-um cmos 8-gb/s 4-pam serial link transceiver. *Journal of Solid-State Circuits*, pages 757–764, May 2000.

[28] M. Farrens, G. Tyson, and A. R. Pleszkun. A study of single-chip processor/cache organizations for large numbers of transistors. In *The 23th Annual International Symposium on Computer Architecture*, pages 338–347, April 1994.

[29] B. G. Fitch, R. S. Germain, M. Mendell, J. Pitera, M. Pitman, A. Rayshubskiy, Y. Sham, F. Suits, W. Swope, T. J. C. Ward, Y. Zhestkov, and R. Zhou. Blue matter, an application framework for molecular simulation on blue gene. *J. Parallel Distrib. Comput.*, 63(7-8):759–773, 2003.

[30] K. Gharachorloo, L. A. Barroso, and A. Nowatzyk. Efficient ecc-based directory implementations for scalable multiprocessors. In *Proceedings of the 12th Symposium on Computer Architecture and High Performance Computing (SBAC-PAD 2000)*, Oct. 2000.

[31] K. Gharachorloo, D. Lenoski, J. Laudon, P. Gibbons, A. Gupta, and J. Hennessy. Memory consistency and event ordering in scalable shared-memory. In *Proceedings of the 17th Int. Symp. on Computer Architecture*, pages 15–26, May 1990.

[32] P. B. Gibbons, M. Merritt, and K. Gharachorloo. Proving sequential consis-

tency of high-performance shared memories. In *Proceedings of the Third ACM Symp. on Parallel Algorithms and Architectures*, pages 292–303, July 1991.

[33] P. N. Glaskowsky. IBM Raises Curtain on Power5. *Microprocessor Report*, Oct. 14 2003.

[34] K. Gniady, B. Falsafi, and T. Vijaykumar. Is SC + ILP = RC? In *Proceedings of the 26th Int. Symp. on Computer Architecture*, pages 162–171, May 1999.

[35] A. Groot, R. Deri, R. Haigh, F. Patterson, and S. DiJaili. High-performance parallel processors based on star-coupled wdm optical interconnects. In *Proceedings of MPPOI*, 1996.

[36] S. Gupta, S. W. Keckler, and D. Burger. Technology independent area and delay estimates for microprocessor building blocks. Technical Report 2000-5, Department of Computer Sciences, University of Texas at Austin, April 2000.

[37] J.-H. Ha and T. Pinkston. The speed cache coherence protocol for an optical multi-access interconnect architecture. In *Proceedings of the Second Workshop on Massively Parallel Processing Using Optical Interconnections*, 1995.

[38] E. Hagersten and M. Koster. Wildfire: A scalable path for SMPs. In *Proceedings of the 5th International Symposium on High-Performance Computer Architecture*, pages 172–181, Jan. 1999.

[39] L. Hammond, B. A. Hubbert, M. Siu, M. K. Prabhu, M. Chen, and K. Olukotun. The stanford Hydra CMP. *IEEE Micro*, pages 71–84, December 2000.

[40] L. Hammond, V. Wong, M. Chen, B. D. Carlstrom, J. D. Davis, B. Hertzberg, M. K. Prabhu, H. Wijaya, C. Kozyrakis, and K. Olukotun. Transactional mem-

ory coherence and consistency. In *Proceedings of the 31st Annual International Symposium on Computer Architecture*, page 102, Jun 2004.

[41] M. Herlihy and J. E. B. Moss. Transactional memory: Architectural support for lock-free data structures. In *Proceedings of the 20th Annual International Symposium on Computer Architecture*, pages 289–300, May 1993.

[42] M. D. Hill, J. R. Larus, S. K. Reinhardt, and D. A. Wood. Cooperative shared memory: software and hardware for scalable multiprocessors. *ACM Transactions on Computer Systems*, 11(4):300–318, 1993.

[43] M. D. Hill and A. J. Smith. Evaluating associativity in cpu caches. *IEEE Transactions on Computers*, 38(12):1612–1629, Dec. 1989.

[44] J. Huh, J. Chang, D. Burger, and G. S. Sohi. Coherence decoupling: Making use of incoherence. In *International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, Oct. 2004.

[45] IEEE. IEEE Standard for Scalable Coherent Interface (SCI), 1992. IEEE 1596-1992.

[46] R. Iyer. CQoS: a framework for enabling QoS in shared caches of cmp platforms. In *Proceedings of the 18th annual international conference on Supercomputing*, pages 257–266, 2004.

[47] N. P. Jouppi. Improving direct-mapped cache performance by the addition of a small fully-associative cache and prefetch buffers. In *Proceedings of the 17th annual international symposium on Computer Architecture*, pages 364–373, 1990.

[48] R. Kalla, B. Sinharoy, and J. M. Tendler. IBM Power5 Chip: A dual-core multithreaded processor. *IEEE Micro*, 24(2), Mar/Apr 2004.

[49] S. Kaxiras and J. R. Goodman. Improving CC-NUMA performance using instruction-based prediction. In *Proceedings of the 5th Int. Symp. on High Performance Computer Architecture*, pages 161 – 170, Jan. 1999.

[50] S. Kaxiras and C. Young. Coherence communication prediction in shared-memory multiprocessors. In *Proceedings of the 6th Int. Symp. on High Performance Computer Architecture*, pages 156–167, Feb. 2000.

[51] R. Kessler. The Alpha 21264 microprocessor. *IEEE Micro*, 19(2):24–36, March/April 1999.

[52] R. Kessler, R. Jooss, A. Lebeck, and M. Hill. Inexpensive implementations of set-associativity. In *Proceedings of the 16th Annual International Symposium on Computer Architecture*, pages 131–139, May 1989.

[53] C. Kim, D. Burger, and S. W. Keckler. An adaptive, non-uniform cache structure for wire-delay dominated on-chip caches. In *Proceedings of the 10th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pages 211–222, October 2002.

[54] S. Kim, D. Chandra, and Y. Solihin. Fair cache sharing and partitioning in a chip multiprocessor architecture. In *Proceedings of the 13th International Conference on Parallel Architecture and Compilation Techniques (PACT'04)*, pages 111–122, 2004.

[55] J. Kuskin, D. Ofelt, M. Heinrich, J. Heinlein, R. Simoni, K. Gharachorloo,

J. Chapin, D. Nakahira, J. Baxter, M. Horowitz, A. Gupta, M. Rosenblum, and J. Hennessy. The Stanford FLASH multiprocessor. In *Proceedings of the 21st Int. Symp. on Computer Architecture*, pages 302–313, Apr. 1994.

[56] A.-C. Lai and B. Falsafi. Memory sharing predictor: The key to a speculative coherent DSM. In *Proceedings of the 26th Int. Symp. on Computer Architecture*, pages 172 – 183, May 1999.

[57] A.-C. Lai and B. Falsafi. Selective, accurate, and timely self-invalidation using last-touch prediction. In *Proceedings of the 27th Int. Symp. on Computer Architecture*, pages 139–148, June 2000.

[58] A. R. Lebeck and D. A. Wood. Dynamic self-invalidation: Reducing coherence overhead in shared-memory multiprocessors. In *Proceedings of the 22nd Int. Symp. on Computer Architecture*, pages 48–59, June 1995.

[59] K.-F. Lee, H.-W. Hon, and R. Reddy. An overview of the SPHINX speech recognition system.

[60] B. E. Lemoff, M. E. Ali, G. Panotopoulos, G. M. Flower, B. Madhavan, A. Levi, and D. W. Dolfi. MAUI: Enabling fiber-to-the-process with parallel multiwavelength optical interconnects. *Journal of Lightwave Technology 22*, 2004.

[61] D. Lenoski, J. Laudon, K. Gharachorloo, W.-D. Weber, A. Gupta, J. Hennessy, M. Horowitz, and M. Lam. The Stanford DASH Multiprocessor. *IEEE Computer*, 25(3):63–79, Mar. 1992.

[62] K. M. Lepak and M. H. Lipasti. Silent stores for free. In *Proceedings of the 33rd Int. Symp. on Microarchitecture*, pages 22–31, Dec. 2000.

[63] K. M. Lepak and M. H. Lipasti. Temporally silent stores. In *Proceedings of the 10th Int. Conf. on Architectural Support for Programming Languages and Operating Systems*, pages 30–41, Oct. 2002.

[64] C. Liu, A. Sivasubramaniam, and M. Kandemir. Organizing the last line of defense before hitting the memory wall for cmps. In *Proceedings of the 10th International Symposium High Performance Computer Architecture*, Feb. 2004.

[65] A. Louri and A. K. Kodi. An optical interconnection network and a modified snooping protocol for the design of large-scale symmetric multiprocessors (smps). *IEEE Transactions on Parallel and Distributed Systems*, (12):1093–1104, Dec. 2004.

[66] 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 *Proceedings of the 30th Int. Symp. on Computer Architecture*, pages 206–217, June 2003.

[67] M. M. K. Martin, M. D. Hill, and D. A. Wood. Token coherence: Decoupling performance and correctness. In *Proceedings of the 30th Int. Symp. on Computer Architecture*, pages 182–193, June 2003.

[68] M. M. K. Martin, M. D. Hill, and D. A. Wood. Token coherence: decou-

pling performance and correctness. In *Proceedings of the 30th Int. Symp. on Computer Architecture*, pages 182–193, June 2003.

[69] M. M. K. Martin, D. J. Sorin, H. W. Cain, M. D. Hill, and M. H. Lipasti. Correctly implementing value prediction in microprocessors that support multithreading or multiprocessing. In *Proceedings of the 34th Int. Symp. on Microarchitecture*, pages 328–337, Dec. 2001.

[70] M. M. K. Martin, D. J. Sorin, M. D. Hill, and D. A. Wood. Bandwidth adaptive snooping. In *Proceedings of the 8th International Symposium on High Performance Computer Architecture (HPCA)*, Feb. 2002.

[71] J. F. Martínez and J. Torrellas. Speculative synchronization: Applying thread-level speculation to explicitly parallel applications. In *Proceedings of the 10th Int. Conf. on Architectural Support for Programming Languages and Operating Systems*, pages 18–29, Oct. 2002.

[72] E. McLellan. The Alpha AXP architecture and 21064 processor. *IEEE Micro*, 13(3):36–47, June 1993.

[73] F. Mederer, I. Ecker, J. Joos, M. Kicherer, H. J. Unold, K. J. Ebeling, M. Grabherr, R. Jager, R. King, , and D. Wiedenmann. High performance selectively oxidized VCSELs and arrays for parallel high-speed optical interconnects. *IEEE Transactions on Advanced Packaging*, 24(4):442–429, Nov. 2001.

[74] A. Moshovos. Regionscout: Exploiting coarse grain sharing in snoop-based coherence. In *Proceedings of the 32nd International Symposium on Computer Architecture*, June 2005.

[75] A. Moshovos, G. Memik, B. Falsafi, and A. N. Choudhary. JETTY: Filtering snoops for reduced energy consumption in SMP servers. In *HPCA*, pages 85–96, 2001.

[76] A. I. Moshovos, S. E. Breach, T. Vijaykumar, and G. S. Sohi. Dynamic speculation and synchronization of data dependences. In *Proceedings of the 24th Int. Symp. on Computer Architecture*, pages 181–193, June 1997.

[77] F. Mounes-Toussi and D. J. Lilja. The potential of compile-time analysis to adapt the cache coherence enforcement strategy to the data sharing characteristics. *IEEE Transactions on Parallel and Distributed Systems*, 6(5):470–481, May 1995.

[78] B. Mukherjee. WDM-based local lightwave networks–part i: Singlehop systems. *IEEE Net. Mag.*, May 1992.

[79] S. S. Mukherjee and M. D. Hill. Using prediction to accelerate coherence protocols. In *Proceedings of the 25th Int. Symp. on Computer Architecture*, pages 179–190, June 1998.

[80] B. A. Nayfeh, L. Hammond, and K. Olukotun. Evaluation of design alternatives for a multiprocessor microprocessor. In *The 23th Annual International Symposium on Computer Architecture*, pages 67–77, May 1996.

[81] B. A. Nayfeh, K. Olukotun, and J. P. Singh. The impact of shared-cache clustering in small-scale shared-memory multiprocessors. In *Proceedings of the 2nd IEEE Symposium on High-Performance Computer Architecture (HPCA)*, page 74, 1996.

[82] A.-T. Nguyen, M. Michael, A. Sharma, and J. Torrellas. The augmint multiprocessor simulation toolkit for intel x86 architectures. In *Proceedings of 1996 International Conference on Computer Design*, Oct. 1996.

[83] O. O. Ogunsola, A. Benner, and J. D. Meindl. A practical symmetric multiprocessor architecture design study using optical multi-drop networks. In *Proceedings of the 5th Annual Austin CAS Conference*, Feb. 2004.

[84] V. S. Pai, P. Ranganathan, S. V. Adve, and T. Harton. An evaluation of memory consistency models for shared-memory systems with ILP processors. In *Proceedings of the 7th Int. Conf. on Architectural Support for Programming Languages and Operating Systems*, pages 12–23, Oct. 1996.

[85] R. Patel, S. Bond, M. Pocha, M. Larson, H. Garrett, R. Drayton, H. Petersen, D. Krol, R. Deri, and M. Lowry. Multiwavelength parallel optical interconnects for massively parallel processing. *IEEE Journal of Selected Topics in Quantum Electronics*, (2), 2003.

[86] Y. N. Patt, W. M. Hwu, and M. Shebanow. HPS, a New Microarchitecture: Rationale and Introduction. In *Proceedings of the 18th Annual Workshop on Microprogramming*, pages 103–108, 1985.

[87] R. Rajwar and J. Goodman. SimpleMP multiprocessor simulator, 2000. Unpublished document.

[88] R. Rajwar and J. R. Goodman. Speculative lock elision: Enabling highly concurrent multithreaded execution. In *Proceedings of the 34th Int. Symp. on Microarchitecture*, pages 294–305, Dec. 2001.

[89] R. Rajwar and J. R. Goodman. Transactional lock-free execution of lock-based programs. In *Proceedings of the 10th Int. Conf. on Architectural Support for Programming Languages and Operating Systems*, pages 5–17, Oct. 2002.

[90] A. Raynaud, Z. Zhang, and J. Torrellas. Distance-adaptive update protocols for scalable shared-memory multiprocessors. In *Proceedings of the 2nd Int. Symp. on High-Performance Computer Architecture*, pages 323–334, Feb. 1996.

[91] S. K. Reinhardt, J. R. Larus, and D. A. Wood. Tempest and Typhoon: User-level Shared Memory. In *Proceedings of the 21st Int. Symp. on Computer Architecture*, pages 325–336, Apr. 1994.

[92] G. Reinman and N. Jouppi. Extensions to cacti, 1999. Unpublished document.

[93] International technology roadmap for semiconductors. Semiconductor Industry Association, 2005.

[94] P. Shivakumar and N. P. Jouppi. Cacti 3.0: An integrated cache timing, power, and area model. Technical Report 2001-2, HP, Western Research Laboratory, 2001.

[95] SimOS PowerPC: Full system simulator. http://http://www.research.ibm.com/simos-ppc.

[96] G. Sohi and M. Franklin. High-performance data memory systems for superscalar processors. In *Proceedings of the Fourth Symposium on Architectural Support for Programming Languages and Operating Systems*, pages 53–62, Apr. 1991.

[97] G. S. Sohi. Instruction issue logic for high-performance, interruptible, multiple functional unit, pipelined computers. *IEEE Trans. Comput.*, 39(3):349–359, March 1990.

[98] G. S. Sohi. Instruction issue logic for high-performance, interruptible, multiple functional unit, pipelined computers. *IEEE Transaction of Computer*, 39(3):349–359, 1990.

[99] G. S. Sohi, S. Breach, and T. Vijaykumar. Multiscalar processors. In *Proceedings of the 22th Int. Symp. on Computer Architecture*, pages 414–425, June 1995.

[100] E. Speight, H. Shafi, L. Zhang, and R. Rajamony. Adaptive mechanisms and policies for managing cache hierarchies in chip multiprocessors. In *Proceedings of the 32nd annual international symposium on Computer Architecture*, 2005.

[101] P. Stenström, M. Brorsson, and L. Sandberg. Adaptive cache coherence protocol optimized for migratory sharing. In *Proceedings of the 20th Int. Symp. on Computer Architecture*, pages 109–118, May 1993.

[102] G. Suh, S. Devadas, and L. Rudolph. A new memory monitoring scheme for memory-aware scheduling and partitioning. In *Proceedings of the 8th International Symposium High Performance Computer Architecture*, Feb. 2002.

[103] G. E. Suh, L. Rudolph, and S. Devadas. Dynamic partitioning of shared cache memory. *Journal of Supercomputing*, 28(1):7–26, 2004.

[104] J. M. Tendler, S. Dodson, S. Fields, H. Le, and B. Sinharoy. Power4 system microarchitecture. *IBM Journal of Research and Development*, 46(1), 2002.

[105] S. J. Wilton and N. P. Jouppi. An enhanced access and cycle time model for on-chip caches. Technical Report 95/3, Digital Equipment Corporation, Western Research Laboratory, 1995.

[106] S. C. Woo, M. Ohara, E. Torrie, J. P. Singh, and A. Gupta. The splash-2 programs: Characterization and methodological considerations. In *Proceedings of the 22nd International Symposium on Computer Architecture*, pages 24–36, 1995.

[107] S. C. Woo, M. Ohara, E. Torrie, J. P. Singh, and A. Gupta. The SPLASH-2 programs: Characterization and methodological considerations. In *Proceedings of the 22th International Symposium on Computer Architecture*, pages 24–36, Santa Margherita Ligure, Italy, 1995.

[108] Q. Yang, G. Thangadurai, and L. Bhuyan. Design of adaptive cache coherence protocol for large scale multiprocessors. *IEEE Transactions on Parallel and Distributed Systems*, pages 281–293, May 1992.

[109] K. C. Yeager. The MIPS R10000 Superscalar Microprocessor. *IEEE Micro*, 16(2):28–40, Apr. 1996.

[110] T.-Y. Yeh and Y. Patt. Two-level adaptive training branch prediction. In *24th Int. Symp. on Microarchitecture*, pages 51–61, 1991.

[111] E. Yeung and M. Horowitz. A 2.4 gb/s/pin simultaneous bidirectional parallel link with per pin skew compensation. In *IEEE International Solid State Circuits Conference*, pages 256–257, February 2000.

[112] R. J. W. Y.S. Liu, W. Hennessy, P. Piacente, J. J. Rowlette, M. Kadar-Kallen, J. Stack, Y. Liu, A. Peczalski, A. Nahata, and J. Yardley. Plastic vcsel array packaging and high density polymer waveguides for board and backplane optical interconnect. In *Proceedings of Electronic Components and Technology Conference*, pages 999–1005, 1998.

Vita

Jaehyuk Huh was born in Daegu, Korea on July 25th 1973, to Jongsik Huh and Philgab Lee. Leaving his hometown for college education, he entered Seoul National University in 1992. He received a Bachelor of Science degree in Computer Science from Seoul National University in February 1996. After a two-year leave for military service, he entered the graduate program in Computer Science at the University of Texas at Austin. He received a Master of Science degree in December 2000.

Permanent Address: 3600 N. Hills Dr. APT 203

Austin TX 78731

This dissertation was typeset with LaTeX 2_ε by the author.