Latency Reduction Techniques in Chip Multiprocessor Cache Systems

by

Michael Zhang

Bachelor of Science in Electrical Engineering and Computer Science
Massachusetts Institute of Technology, May 1999

Master of Engineering in Electrical Engineering and Computer Science
Massachusetts Institute of Technology, May 1999

Submitted to the Department of Electrical Engineering and Computer Science
in partial fulfillment of the requirements for the degree of

Doctor of Philosophy

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

January 2006
[February 2006]

Author .....
Department of Electrical Engineering and Computer Science
January 20, 2006

Certified by .....
Krste Asanović
Associate Professor
Thesis Supervisor

Accepted by .....
Arthur C. Smith
Chairman, Department Committee on Graduate Students

Latency Reduction Techniques
in Chip Multiprocessor Cache Systems

by

Michael Zhang

Submitted to the Department of Electrical Engineering and Computer Science
on January 20, 2006, in partial fulfillment of the
requirements for the degree of
Doctor of Philosophy

Abstract

Single-chip multiprocessors (CMPs) solve several bottlenecks facing chip designers today. Compared to traditional superscalars, CMPs deliver higher performance at lower power for thread-parallel workloads.

In this thesis, we consider tiled CMPs, a class of CMPs where each tile contains a slice of the total on-chip L2 cache storage, and tiles are connected by an on-chip network. Two basic schemes are currently used to manage L2 slices. First, each slice can be used as a private L2 for the tile. Private L2 caches provide the lowest hit latency but reduce the total effective cache capacity because each tile creates a local copy of any block it touches. Second, all slices are aggregated to form a single large L2 shared by all tiles. A shared L2 cache increases the effective cache capacity for shared data, but incurs longer hit latencies when L2 data is on a remote tile. In practice, either private or shared works better for a given workload.

We present two new policies, victim replication and victim migration, both of which combine the advantages of private and shared designs. They are variants of the shared scheme which attempt to keep copies of local L1 cache victims within the local L2 cache slice. Hits to these replicated copies reduce the effective latency of the shared L2 cache, while retaining the benefits of a higher effective capacity for shared data. We evaluate the various schemes using full-system simulation of single-threaded, multi-threaded, and multi-programmed workloads running on an eight-processor tiled CMP. We show that both techniques achieve significant performance improvement over baseline private and shared schemes for these workloads.

Thesis Supervisor: Krste Asanović
Title: Associate Professor

Acknowledgments

I am extremely fortunate to have Professor Krste Asanović as my Ph.D. adviser. His wealth of knowledge, sheer brilliance, and dedication to his students are far beyond any graduate student could ever ask for. This thesis would not have been possible without his constant help, support, and encouragement. Thank you for mentoring me all these years, Krste.

I want to thank my thesis readers, Professor Srini Devadas and Professor Larry Rudolph, for all their feedback on this thesis. I also want to thank Professor Anant Agarwal, Professor Charles Leiserson, and Professor Arvind for all their advice.

Fundings for my graduate work came from DARPA HPCA/PERCS project W0133890, CMI project 093-P-IRFT, NSF CAREER Award CCR-0093354, DARPA Award F30602-00-2-0562, and donations from Intel Corporation and Infineon Technologies.

Many thanks to members of the SCALE group, Elizabeth Basha, Chris Batten, Jae Lee, Rose Liu, and Emmett Witchel. Special thanks to all my co-authors, Ronny Krashinsky, Seongmoo Heo, Albert Ma, Luis Villa, Ken Barr, and Heidi Pan. I learned a lot from you guys. Special thanks to my former and current officemates, Mark Hampton, Jessica Tseng, Seongmoo Heo, and Steve Gerding. It's been a pleasure.

Many thanks to all the good friends I made in school, Mark Hampton, Daihyun Lim, Ed Suh, Charlie O'Donnell, Ian Bratt, and Jason Kim.

To the original gang of Team Five (and its various later forms that I shall not name), Mike Gordon, Sam Larsen, Mark Stephenson, Ronny Krashinsky, and Steve Gerding. I want to thank you for all the awesome times that we had together. You made the past seven years that much more enjoyable for me. I also thank you for the many heated yet fun debates on just about every topic a rational person could care for and then some.

To Min Shao and Claire Chen, whom I vented on consistently during my thesis writing days. Thank you for all you patience, advice, and hospitality, especially in the last few months of my Ph.D.

To Mary McDavitt, whom I bother on a daily basis but has always managed to put up with me. Thank you for all your help and the many Redsox vs. Yankees discussions.

To Heidi Pan, you are the source of my happiness and I am grateful I met you.

To my Mom Xiaonan and my Dad Jiawei, without you, I wouldn't be where I am today. Your unconditional love for me is beyond any words can describe. Thank you.

Contents

List of Figures

List of Tables

Chapter 1

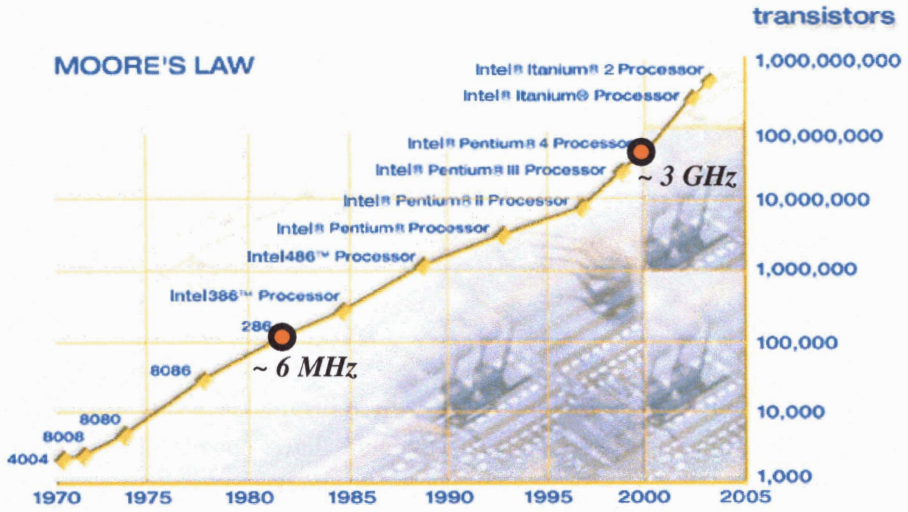
Introduction

Over the past two decades, VLSI technology advances have closely followed Moore's Law. From the mid 1980's, microprocessor clock frequencies have increased by over 100X and on-chip capacities have increased by over 10,000X, as shown in Figure 1-1(a). These two technological improvements have led to a period of rapid performance growth for general-purpose microprocessor systems. During this time, highly sophisticated microprocessors such as the Intel Pentium4 [HSU+01] and Alpha 21264 [Kes99] have been built, featuring clock frequencies reaching several gigahertz, deep pipelines, large caches, and numerous performance-enhancing microarchitectural features.

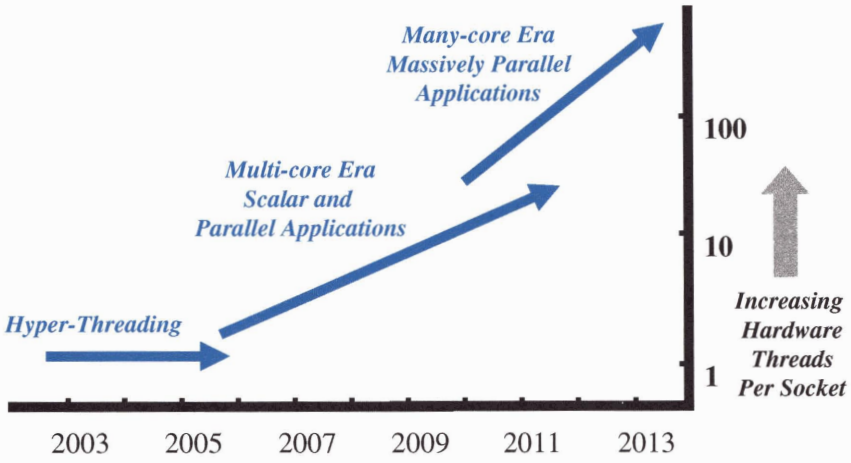
Despite the success of wide superscalars, we are in the midst of a drastic architectural design paradigm shift. The ten-year industry outlook in Figure 1-1(b) shows that the design focus has shifted to *single-chip multiprocessors*, which place multiple replicated uniprocessor cores onto the same die, instead of more aggressive optimizations of uniprocessors [Kre04a, KST04, KMAC03, CR05, KAO05, Raz05, Cav05]. Table 1.1 summarizes the main features of some current CMPs.

This thesis investigates various design alternatives to improve the performance and reduce the power consumption of the on-chip cache system in these CMP architectures. Compared to previous uniprocessor cache systems, CMP caches have two distinct features that present new challenges. First, the size of the on-chip cache will continue to grow, creating the phenomenon of *non-uniform access latency (NUCA)*. A NUCA architecture allows various parts of the cache to be accessed with different latencies, depending on the physical location. Therefore, a strategic (distance-aware) physical placement of cached data can significantly improve performance. Second, the on-chip cache system must be able to provide low access latencies to multiple on-chip cores simultaneously.

The main contributions of this thesis are two innovative CMP cache management policies: *victim replication* and *victim migration*. These two techniques achieve significant reductions on cache fetch latency and communication power over the baseline private and shared designs. They are simple to implement and provide robust performance over a wide range of applications.



(a)



Source: www.intel.com/technology/silicon/power/multicore.htm

(b)

Figure 1-1: (a) Intel's products have closely followed Moore's Law. Clock frequencies have increased over 100X and on-chip transistor counts have increased over 10,000X in the last 25 years. (b) Intel processor technology road map for the next ten years. The number of processor cores is expected to reach into the hundreds by early next decade.

	Year	Cores (Hardware Threads per Core)	Tech. (nm)/ Transistor #/ Freq. (GHz)	Inter-connect	L2 Cache Configuration size/assoc/ latency	L2 Cache Sharing Pattern
Server Processors						
IBM Power5	2003	2(2)	130/276M/1.9	Bus	1.9MB/10/13	Shared
AMD Opteron	2004	2(1)	90/233M/2.2	Bus	1MB/16/12	Private
Intel Montecito	2005	2(2)	90/1.7B/1.8	Bus	24MB/12/14	Private
Sun Niagara	2005	8(4)	90/N.A./N.A.	Bus	3MB/8/N.A.	Shared
Embedded Processors						
RMI XLR	2005	8(4)	90/N.A./1.5	Ring	2MB/8/N.A.	Shared
Cavium Octeon	2005	16(1)	90/N.A./0.6	Bus	1MB/N.A./N.A.	Shared
SiByte BCM14xx	2005	4(1)	90/N.A./1.2	N.A.	1MB/N.A./N.A.	Shared

Table 1.1: Comparisons of several leading industry CMPs. These CMPs show the trends of higher processor core counts, increased outer-level cache capacities, and moderate clock frequencies.

1.1 Why CMPs? Why Now?

Ready or not, we are living in the dawn of *single-chip multiprocessors (CMP)*. The continued performance improvement brought by technological advances, however, has slowed down dramatically in the past four to five years. This slowdown can be attributed to three key factors.

First, more complex microarchitectural designs can only bring marginal performance gain at the expense of significantly higher design efforts and longer design cycles. The traditional channels to improving performance by widening the issue widths and using better speculation mechanisms are fundamentally limited by the amount of *instruction-level parallelism* (ILP) inherent in the workloads. These methods have already reached diminishing returns.

Second, higher clock frequencies can no longer be directly translated into better performance because global wire delay does not scale with the silicon feature size. For each subsequent technology generation, less on-chip area can be reached within one clock cycle, leading to longer cross-chip latencies [HMH01, AHKB00]. Thus, even though individual chip components continue to become faster, the communication latency among different components cannot, limiting the performance of the overall system.

Third, power consumption has become a key design constraint that limits achievable processor performance. In traditional desktop and server systems, power usages exceeding the hundred-watt range require exotic cooling systems. Elevated power density causes transistor reliability and stability problems, and higher die temperature leads to leakier and slower transistors. In the mobile computing arena, power dissipation is directly correlated to battery life, thus to the usability of the mobile device itself. The increasing power usage is the primary factor that finally forced chip designers to deviate, at least temporarily, from evolving traditional superscalar uniprocessors [Kre04a].

1.2 Software Implications

Traditional superscalars and VLIWs exploit *instruction-level parallelism* (ILP), relying on speculative execution to gain performance. Because the instruction-level parallelism that exists in sequential programs is limited, even the most elaborate systems today can only achieve a marginal performance gain with better prediction and speculation mechanisms. CMPs exploit a much coarser form of parallelism at the thread level, which we refer to as *thread-level parallelism* (TLP). For applications with significant TLP, CMPs can deliver higher throughput and consume less energy per operation than a wider-issue superscalar [ONH+96]. Several important classes of applications have abundant thread-level parallelism and can take advantage of CMPs.

1. *Server Workloads*: Large transaction-based server workloads, such as web or database servers, are inherently thread-parallel because each transaction is an independent task. Today, server workloads are executed on large multi-chip multiprocessor systems to obtain high throughput. CMPs will work very well for these workloads.

2. *Parallel Scientific Workloads*: Classic algorithms, such as Fourier transform or LU decomposition, are the centerpieces of many critical scientific workloads. Large compute-intensive programs, such as weather forecasting, demand extremely high performance that uniprocessors are unable to deliver. Because of their importance, they are well studied and heavily parallelized at the thread level to take advantage of large multi-chip systems. These scientific workloads will work even better on CMPs because they have tighter integration that reduces communication latencies among different cores and memory.

3. *Multi-Programmed Workloads*: Most commercial modern operating systems support multitasking and can run a large number of different programs in parallel. In fact, desktop machines today run hundreds of programs concurrently using time-sharing. Thus, we anticipate multi-programmed workloads to be the most common ones for a desktop processor. Multi-programmed workloads are naturally thread-parallel as different programs rarely share data, thus fully utilizing the features of a CMP.

1.3 Hardware Implications

From a hardware point of view, CMPs address three key bottlenecks of unicore processors: (1) power budget; (2) global wire delay; and (3) design complexity.

1. *Power Budget:* CMPs achieve high performance by running different threads in parallel, putting less pressure on individual thread performance. Thus, CMPs can use

relatively less aggressive cores and scale back clock frequency. This approach sacrifices some single-thread performance, but allows many power-inefficient features to be removed from the processor, thereby dramatically reducing energy per operation.

2. *Global Wire Delay:* The physical structure of a CMP naturally constricts the majority of the data movement to be localized within each processor core. Global wires in a CMP will mainly be responsible for transporting shared data between different threads. While increasing global wire delay will remain a problem, such global communication happens much less frequently compared to, for example, accesses to the register file in a wide superscalar. In addition, this abstraction gives more control over the wire delay problem to the software. For example, the operating system can place multiple threads that have a high degree of data sharing in adjacent cores to minimize the cost of global communication.

3. *Design Complexity:* The CMP approach dramatically reduces design complexity by allowing the chip makers to reuse previous core designs with minor modifications to suit future products. The focus of the redesign effort is the interconnection network responsible for communication among cores, caches, physical memory, and I/O devices. Thus CMPs can have a much shorter design cycle and time to market compared to superscalars.

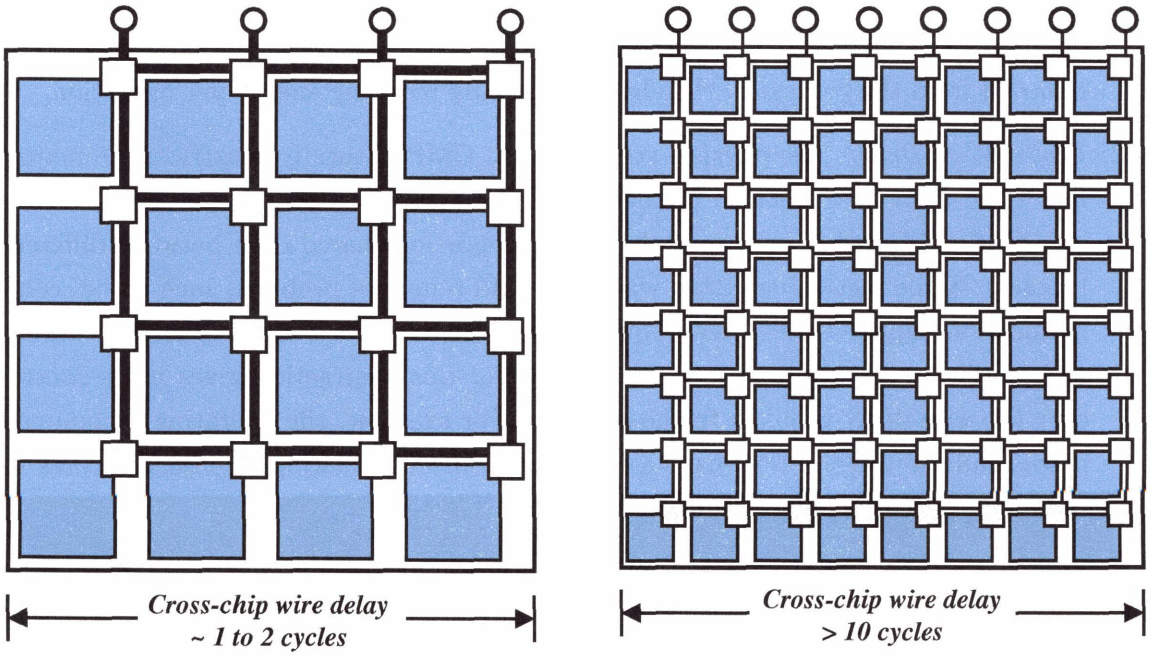
1.4 CMP Design Trends

There are two trends in future CMP designs. First, CMPs will have more cores. For example, the Niagara [KAO05] and the XLR [Raz05] chips have 8 cores and the Cavium Octeon CN38xx chip [Cav05] has 16 cores. Each core is likely to be relatively simple, especially in the embedded chip space. Second, CMPs will have more total cache capacity. For example, the newest Intel Montecito chip, based on the Itanium, has two cores, each with its own 12MB L3 cache, forming a total on-chip capacity of over 24MB [CR05].

1.5 Non-Uniform Access Latency

Most current cache designs divide large caches into small slices to reduce both access latency and energy consumption. The cache access latency is primarily dominated by the access time of each individual cache slice, thus the access latencies to various slices are fixed. We refer to this type of cache as a *uniform cache access (UCA)* cache, as shown in Figure 1-2(a).

In the larger caches anticipated in future CMPs, wire delay will cause cross-chip communications to reach tens of cycles [HMH01, AHKB00]. Cache fetch latencies will be dominated by the wire delay to reach each individual cache slice rather than the time spent accessing the slice itself. The access latencies to various slices will become significantly different depending on their locations with respect to the load/store unit of the processor. UCA design



(a) ***UCA*** has Short cross-chip latency. Data array access time dominates overall cache access time

(b) ***NUCA*** has Long cross-chip latency. Wire delay dominates overall cache access time

Figure 1-2: Each block represents an optimally sized cache slice for power consumption and access latency. (a) Uniform cache access (UCA) used by most current cache designs. (b) The non-uniform cache access (NUCA) anticipated in the future cache designs.

is no longer suitable for these wire-dominated caches because using the worst-case latency will result in unacceptable hit times. Thus, we must allow different slices of the cache to be accessed at their fastest possible latencies. The resulting cache design is what we refer to as a *non-uniform cache access (NUCA)* cache [KBK02] as illustrated in Figure 1-2(b).

A NUCA architecture can be either static or dynamic. A static NUCA (S-NUCA) simply relaxes a UCA design and allows different cache slices to be accessed with different latencies. In this case, each cache block is statically mapped to a specific bank.

The more flexible dynamic NUCA (D-NUCA) cache exposes the physical location of each cache block to the designer, allowing more optimal placement than the statically address-mapped approach of S-NUCA. An intelligent placement maps data to physical cache locations such that the working set of the workload stays in the cache slices physically closest to the core. Such a placement minimizes the cross-chip communication latency incurred by cache accesses. However, the process of locating a cache block in a D-NUCA can cost significantly more time and energy than in a S-NUCA.

1.6 Thesis Focus: CMP Data Access Latency

1.6.1 Thesis Problem Statement

For any computer system, its overall performance is often directly correlated to the performance of its memory hierarchy. In future CMPs, off-chip misses will remain expensive, but increases in clock frequency, together with worsening global wire delays, will also increase latencies for cross-chip communication. Effective use of on-chip caches must therefore consider both the cost of off-chip misses and the cost of cross-chip communications. Two baseline outer-level cache designs, *private* and *shared*, illustrate the trade-offs between these two components of effective data access latency. For simplicity, we assume in the rest of the thesis that the second-level cache (L2) is the outer-most level of on-chip cache. A private design evenly partitions all of the on-chip L2 cache slices such that each processor is assigned its closest partition as its private L2 cache. The shared design aggregates all the L2 cache slices to form a single L2 cache shared by all the cores.

The private design has a low L2 hit latency, as the private L2 cache is physically co-located with the processor core and has a much smaller area than a shared cache. This layout provides good performance if the working set fits within the local L2 slice. The disadvantage of the private L2 design is that effective on-chip cache capacity is reduced for shared data, as each core must retain its own copy of any shared data block. The shared design reduces the off-chip miss rate for large shared working sets because only a single on-chip L2 cache copy is required for any shared data. However, large shared L2 caches have a worse access latency than a small private L2 cache.

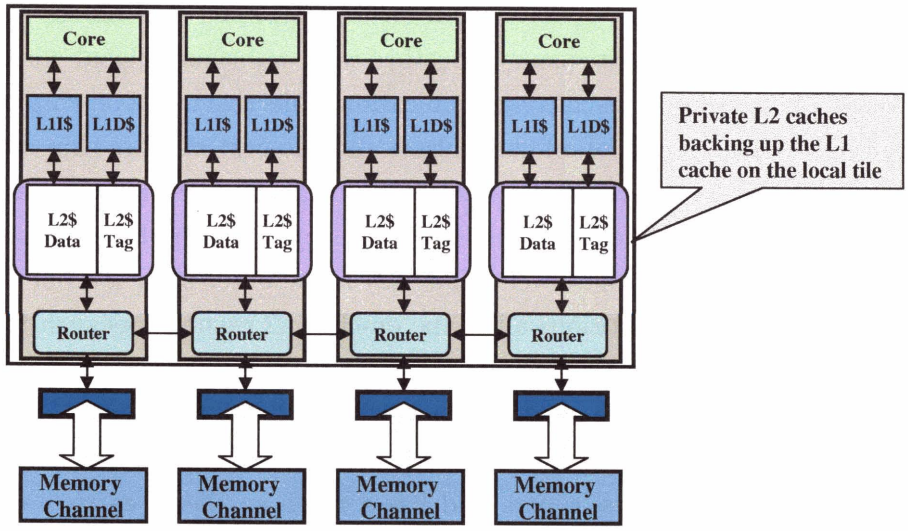
With multiple cores, this placement task becomes particularly challenging because many cores may contend for the same shared data simultaneously. The optimal placement of the shared data may not be close to any of the requesting cores, thus making them unable to provide fast access time to most of the sharers.

In this thesis, we will investigate various cache management policies of cache hierarchies in CMPs. We study the private and shared cache designs described above and explore novel cache management schemes with optimal trade-offs between the off-chip miss rate and the cross-chip latency to achieve lower data access latencies for future CMPs.

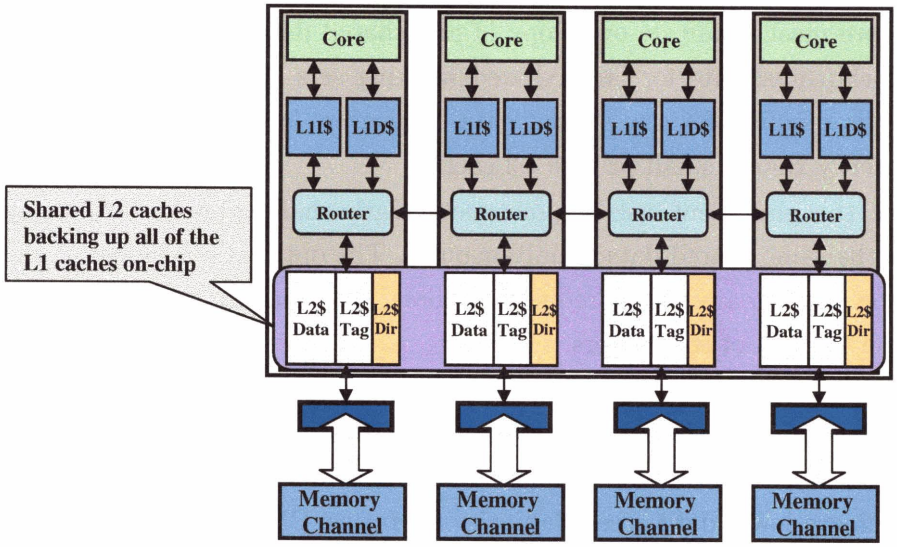
1.7 Thesis Outline

Even though CMPs are a relatively new architectural design target, they are closely related to earlier multi-chip multiprocessor systems. Chapter 2 provides the necessary background for these earlier systems, and draws parallels between distributed shared memory systems (DSMs) and CMPs. Cache coherence protocols are briefly introduced. We also discuss various pertinent latency-hiding techniques used in DSM systems.

Chapter 3 describes our take on future CMPs trends, which we believe will naturally



(a) Private design for L2 caches



(b) Shared design for L2 caches

Figure 1-3: The two baseline L2 cache designs. (a) The private design evenly partitions all of the on-chip L2 cache capacity such that each processor is assigned its closest partition as its private L2 cache. (b) The shared design aggregates all the L2 cache capacity to form a single L2 cache shared by all the cores.

evolve toward arrays of replicated tiles connected over a switched network. We call this architecture a *tiled CMP* and use it as the basis for the thesis. We then present the implementation of the private design and the shared design on a tiled CMP, and discuss various design issues and overhead.

Chapter 4 describes two novel approaches, *victim replication* [ZA05b] and *victim migration* [ZA05a], which combine the advantage of private and shared designs to reduce both the off-chip miss rate and the cross-chip access latency. We present the implementation of these two techniques as well as associated cache replacement policies to manage these two architectures.

Chapter 5 describes the experimental methodology used in this thesis. We describe the processor and cache simulator used, as well as their integration. The workloads chosen to evaluate the designs are presented. The effects of fastforwarding and system variability are also discussed. The experimental results are presented in Chapter 6. They show that the latency reduction techniques proposed by our research are robust, performing well for a wide range of workloads.

Chapter 7 summarizes this thesis and highlights our contributions. In addition, we point out some of the limitations this thesis had in evaluating the effectiveness of the cache designs, ending with a discussion of designing cache and memory systems for the massive CMPs anticipated in the future. Finally, in Appendix A, we give an overview of the cache coherence protocol used in this thesis.

1.8 Glossary

To facilitate the discussion in the rest of this thesis, we use the following abbreviated terms to describe the various architectures or systems presented in this thesis.

1. *Unicore Architecture*: A microprocessor architecture with only one core on chip. Most existing microprocessors belong to this category.

2. *Uniprocessors*: Synonymous with unicore architecture.

3. *Multicore Architecture*: A microprocessor with a moderate number (more than one) of cores. All of today's CMP architectures belong to this category.

4. *Manycore Architecture*: A microprocessor architecture with a large number of cores on-chip. We anticipate seeing these architectures in the future.

5. *CMP*: Single-chip multiprocessors. Synonymous to multicore or manycore architectures. In this thesis we only consider symmetric CMPs, i.e., all cores are functionally identical.

6. *Multi-chip Multiprocessor systems*: A system that consists of multiple uniprocessors. Earlier multiprocessor systems all belong to this category.

7. *Multi-chip CMP systems*: A system that consists of multiple CMPs, such as the AMD Opteron system.

8. *Wide Superscalars*: We collectively call advanced unicore microprocessors with wide issue width, deep pipeline and sophisticated microarchitectural features "wide superscalars". Examples include the Intel Pentium 4 and the Alpha 21264.

Chapter 2

Multiprocessing Background

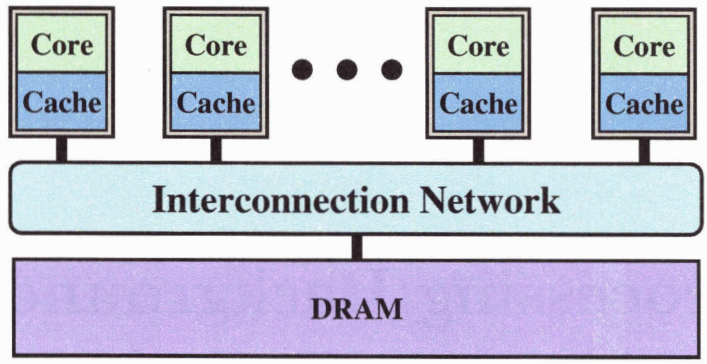
Chip multiprocessors are closely related to earlier multi-chip multiprocessor systems. In this chapter, we discuss the similarities and differences between these two systems by reviewing some basics of the multi-chip multiprocessor systems, including cache coherence, an important and necessary component in both systems. Moreover, we discuss non-uniform memory accesses and present related work in latency-reduction for memory data accesses in multi-chip systems [Bur92, HLH92, ACJ+99, LLG+92, KOH+94, GW94, Cor91a, Inc93, Cor93, Cor91b, CYS+93]. While these earlier techniques target multi-chip system memory accesses, future CMPs will present similar problems due to their NUCA memory systems. Thus, understanding these techniques can help us to devise appropriate latency-reduction techniques for CMP NUCA caches.

2.1 Multi-Chip Multiprocessor Systems

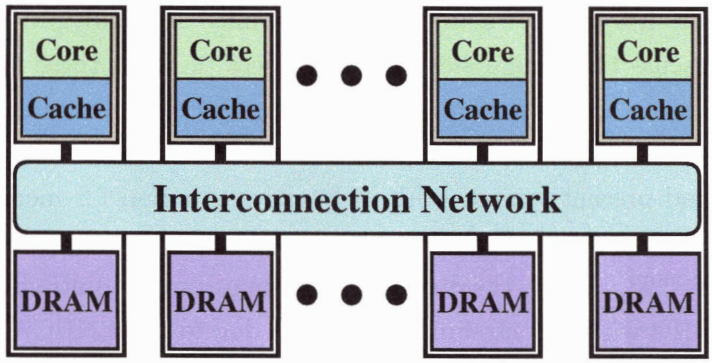
Traditionally, multiprocessor systems have been primarily used to run server and scientific workloads. They are constructed by interconnecting multiple uniprocessors and DRAM modules. Compared to uniprocessors, a multi-chip system is capable of delivering computing power that is several magnitudes higher. However, these workloads must be carefully written and tuned to contain a high degree of thread-level parallelism that can be efficiently exploited by a multiprocessor.

2.1.1 Multiprocessor Memory Hierarchy Layout

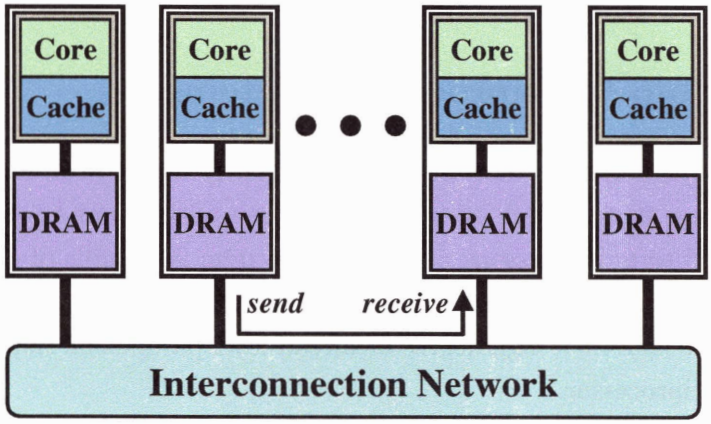
The design and performance of the memory system directly affects the overall system. Figure 2-1 shows three alternatives, differing in the way they each store and access data. Figure 2-1(a) shows a physically centralized memory shared by all of the processors, interconnected through a shared bus. While this approach is simple, it can only be applied when the number of processors in the system is small. Larger multi-chip systems generally have hundreds of processors and the bandwidth of a physically centralized memory system does



(a) Physically Centralized Memory



(b) Physically Distributed Shared Memory



(c) Physically Distributed Message Passing

Figure 2-1: Distribution schemes for multi-chip multiprocessors. (a) *Physically centralized memory*: Used in smaller systems where the centralized memory can be shared by all nodes and provide a reasonable latency and bandwidth. (b) *Physically distributed shared memory system*: Used in larger systems; memory is physically (evenly) distributed to reduce fetch latency and improve memory bandwidth. For case (a) and (b), a coherence protocol is required to keep cached data coherent. (c) *Physically distributed message passing system*: Each memory module is private to its co-located processor. Software generates explicit messages to transport shared data among different nodes.

not scale with the processor count. In these large multi-chip systems, physical memory is typically distributed across the system, with a portion of the memory co-located with each processor. A communication protocol is used to manage the exchange of shared data between different processors. This approach is illustrated in Figure 2-1(b) and Figure 2-1(c). Traditionally, designers have taken two approaches to implementing a physically distributed memory system: *message passing* and *distributed shared memory*, described below.

2.1.2 Message Passing

From a hardware standpoint, a message passing system is equivalent to a multi-computer system with many independent computers tightly integrated through a high-bandwidth interconnect. Each node in the system has its own processor, local cache, and associated memory module. Each memory module is private to the local node and has its own address space that cannot be seen by any remote nodes. In order to share data in a message passing system, the operating system must provide a set of user-level communication primitives or protocols with *send* and *receive* commands. Software must explicitly specify the data communication among the various processors. Because software handles the complexity of data sharing in a message-passing system, the underlying hardware becomes straightforward to build.

The memory-fetch latency in a message passing system is short because the memory is local to the processor. However, since each *send* or *receive* message is handled in software, the inter-node communication latency is very high. Therefore, message passing systems work very well for workloads that have little data sharing among threads, because they require minimal amounts of communication among various nodes. Some early message-passing systems include the Intel Paragon XP/S [Cor91a] and the CM-5 [Cor91b] from Thinking Machines. Common message-passing systems today are generally cluster systems often with custom high-performance interconnect, such as the IBM SP2 clusters.

2.1.3 Distributed Shared Memory

An alternative to message passing is the *distributed shared memory* (DSM) approach. In a DSM system, all of the physically distributed memory modules are combined to form a logically unified address space shared by all nodes. A data block is stored in the memory module of its *home node*, which is usually statically determined by its address. Data sharing among different processors is implicit as each processor simply issues loads and stores to the unique address of the shared data. Compared to message passing, the shared memory model removes the need for programmers to explicitly direct the shared data movement in the system. In addition, multi-threaded programs written for sequential machines can be easily ported over to a DSM machine.

Traditionally, DSM systems are often referred to as *non-uniform memory access* (NUMA) machines, because the latency of a memory access is dependent on the relative locations

of the requesting processor and the memory module hosting the requested data. If they happen to be on the same node, then we refer to the access as a *local access*. On the other hand, if they are located on different nodes, we refer to the access as a *global access* or a *remote access*. Global accesses generally take much longer and the exact latency depends on a number of other factors such as the network latencies and congestion.

Because memory is shared by all nodes in a DSM and each node may choose to cache shared data locally, care must be taken to ensure that all nodes have a consistent view of the memory content. Specifically, each load to a memory location must see the value committed by the last store to the same location. This property is referred to as *cache coherence* [CF78]. In the presence of caches, this property can be easily violated because each processor can store a locally cached copy of the data that may be newer than the copy stored in memory. DSM systems typically use hardware protocols to ensure cache coherence, giving the programmer a simple and coherent view of memory from all threads. Cache coherence is a well-studied field, and we briefly review basic protocols in Section 2.3. Some early DSM machines include the SGI Challenge [GW94], the Cray T3D [Inc93], and the KSR-1 [Bur92] from Kendall Square Research.

2.2 CMP Systems versus DSM Systems

The first CMPs closely resemble a tight integration of earlier multi-chip multiprocessor systems. Figure 2-2 shows a generic physical layout of an eight-node CMP. A centralized on-chip network ties together eight cores with their small private L1s and a large shared outer-level (L2) cache. This type of layout, which we refer to as the "dance-hall" layout, is quite common among current commercial CMPs, such as the Niagara processor from Sun Microsystems [KAO05] and the XLR processor from Raza Electronics [Raz05].

The main difference between an earlier DSM system and a modern CMP system lies in the communication network. Communication delay between two nodes in a generic DSM system can take hundreds of cycles because messages travel through an *inter-chip* network. Off-chip operations generally are clocked at a fraction of the chip frequency, and are limited by on-chip pin bandwidth. In a CMP, however, communication between processor cores travels through an *on-chip* network, which can deliver much higher bandwidth at lower latencies, significantly lowering the cost of inter-node communication compared to DSMs.

2.3 Cache Coherence Protocols

Cache coherence makes sure all of the processors in shared-memory systems have consistent views of the memory, a necessary and important component for program correctness and performance. A coherence mechanism typically has two components: (1) storage holding data sharing information; and (2) a set of protocols that maintains data coherence using

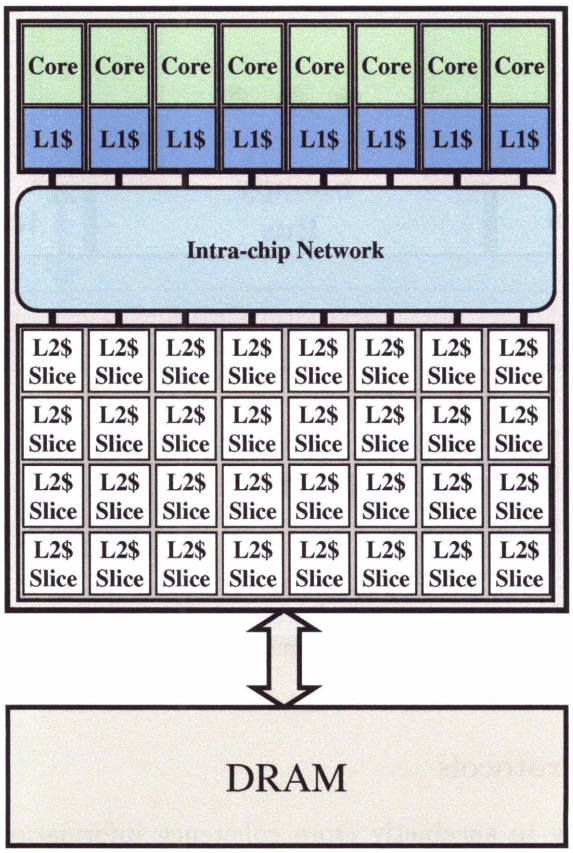


Figure 2-2: Current CMPs resemble tightly-integrated versions of a multi-chip multiprocessor system of the 1980s. Processor cores are tightly coupled with the L1 caches, and connected by a centralized high-bandwidth, on-chip communication network to large outer-level caches.

the sharing information. Therefore, when a processor accesses a data block, the protocol performs two essential tasks. First, it determines the location and the status of all the cached copies of the requested block. Second, it updates the status and/or data of these copies accordingly.

The status of the cached copies of any block is usually kept by attaching state to each cache data block. The simplest cache coherence protocol categorizes a cache block into one of three states: (1) the *invalid* (or *I*) state, means that the cache block is not holding valid data; (2) the *shared* (or *S*) state, means that the block is shared by one or more processor caches in the system. Shared blocks can only be read from, but not written to, and the value held in the block is identical to the copy held in memory; and (3) the *modified* (or *M*) state, means that the block is uniquely held. We call this node the *owner* of the block, and it has the right to modify. Because the owner may hold newer data than memory, it must write any evicted cache blocks back to memory. This protocol is commonly referred to as the *three-state MSI* protocol. More sophisticated protocols employ additional states to reduce coherence traffic as well as fetch latency. Two popular protocols are *MESI* [PP86, AB86] and *MOESI* [SS86].

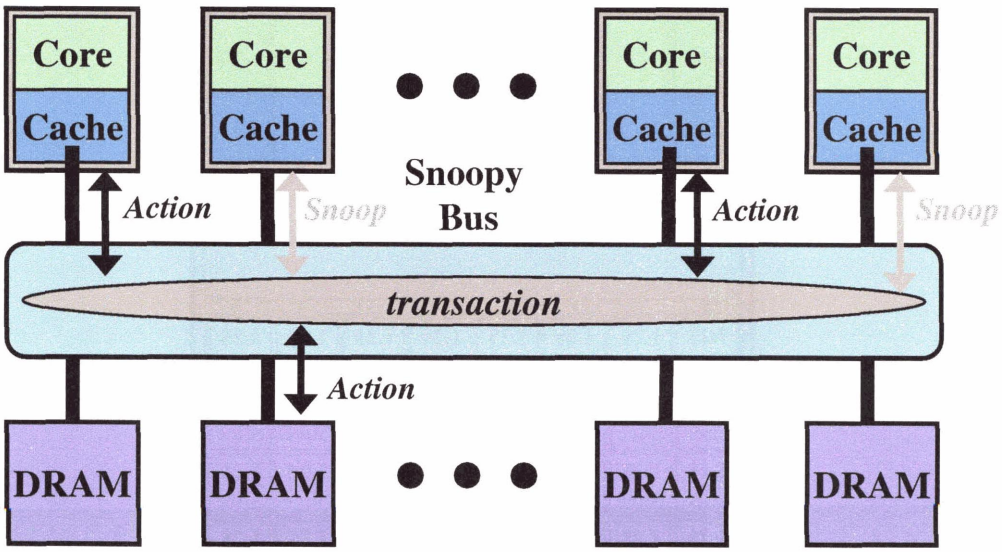


Figure 2-3: Illustration of a snoopy bus-based protocol. When a coherence transaction message is placed on the bus, all of the caches and DRAM modules snoop the message, but only the relevant parties take the appropriate actions.

2.3.1 Bus-based Protocols

Now that we know how to succinctly store coherence information, we must be able to retrieve it and take appropriate action. The bus-based approach, a simple technique first proposed by James Goodman in 1983 [Goo83], uses a snoopy bus shared by all the nodes in the system. Each node has a cache controller and a memory controller that monitors, or *snoops*, the transactions on the bus. The relevant parties involved in a transaction take appropriate action, as shown in Figure 2-3. This protocol is simple to implement and can be applied to all multi-chip systems that use a shared bus to connect the nodes in the system.

During a cache load miss, the requesting cache places a load request onto the bus. All caches and memory modules snoop the request to determine whether they should take any action. If the requested block is held in a shared state by the memory module at the home node, the data is placed onto the bus and snooped by the requestor, completing the transaction. If data is held in a modified state, the owner cache downgrades to a shared state and places the modified data onto the bus, which is snooped by both the requestor and the home node, completing the transaction. Store miss works similarly, except that all the cached copies of the requested block must be invalidated and written back if dirty.

2.3.2 Directory-Based Protocols

While elegant and simple to implement, the applicability of bus-based protocols is limited by the system's ability to provide a fast shared bus. Since the sharing information is kept at each cache and memory module in a decentralized fashion, all nodes must snoop every

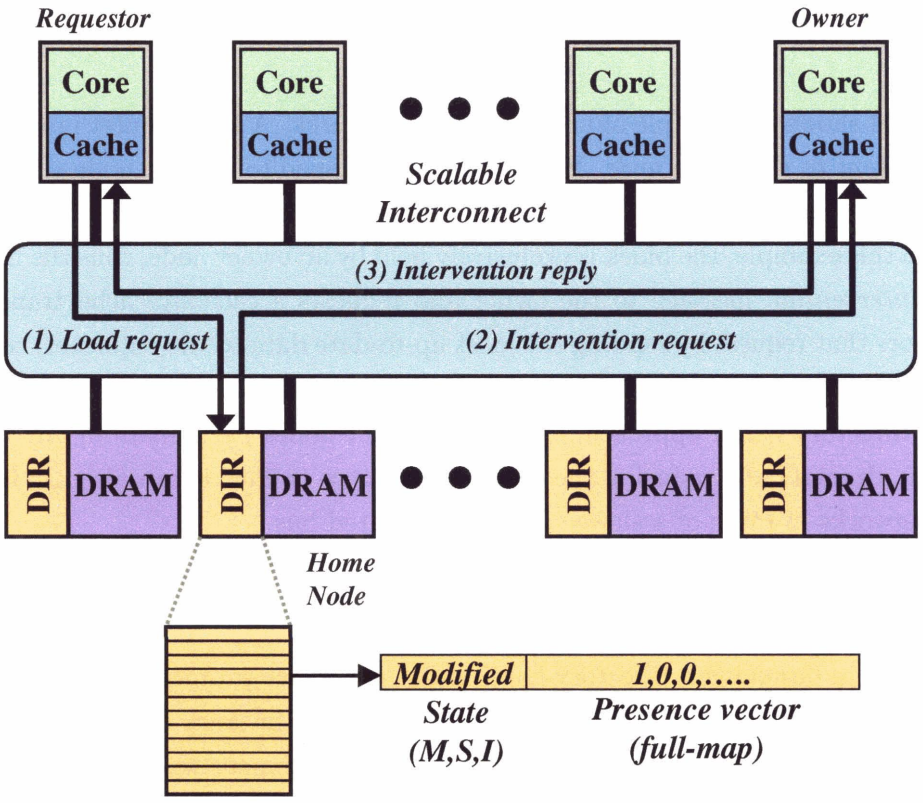


Figure 2-4: Illustration of a directory-based protocol. When a cache miss initiates a coherence transaction, the request message is sent to its home node (generally determined statically by the requesting address). The home node holds the directory entry with all of the relevant sharing information of the requested block.

bus transaction, even when only a small percentage of the nodes in the system are involved in a particular coherence transaction. Thus, the effectiveness of the coherence protocol is dictated by the bandwidth and latency of the bus broadcast operation. As the node count scales up, the broadcast operation will no longer be able to reach all nodes within a reasonable time.

Directory-based protocols are designed to combat the bandwidth limitation of the bus-based protocol, by breaking an expensive broadcast into a sequence of point-to-point messages that only involve the relevant parties in any transaction. Therefore, the protocol no longer requires all nodes to share a common bus and snoop the transaction, but rather calls for an interconnection network that can efficiently transport messages among different nodes. In order to quickly identify the relevant parties in the transaction, the directory-based approach logically centralizes sharing information into a *directory*, as shown in Figure 2-4. The directory is usually co-located with the data block in memory, with one directory entry corresponding to one memory block. Each directory entry keeps two pieces of essential information about the block, its state, and the *presence vector*. Together they track the block's current sharers and their read and write privileges. The simplest presence

vector uses one bit for each node in the system, which is commonly referred to as a *full-map* vector.

Figure 2-4 shows an example of how the directory works. A load request initiated by the requestor is sent to the *home node*. Each address is mapped to a home node statically, and the home node has the directory with the state and the presence vector of the requested address. In this example, the block is exclusively held by an owner node, thus the home node sends an *intervention* message to the owner and requests a cache-to-cache transfer. The owner honors that request by sending the most up-to-date data to the requestor, completing the transaction.

In the directory-based approach, only the relevant nodes participate in any coherence transaction, dramatically reducing the on-chip coherence traffic. It can also use fast point-to-point networks in place of a slower monolithic shared bus.

Managing Directory Size

One challenge in designing a directory-based cache coherence protocol is the directory size management in systems with high node counts because the area overhead caused by a full-map directory (one bit per node) can be prohibitively expensive.

The *limited directory protocol* was proposed by [ASHH88], in which each directory entry only holds up to a fixed number of sharers. When the actual number of sharers exceeds the maximum, current sharers are evicted in favor of new sharers. This technique is based on the observation that on average, only a small fraction of the overall nodes are involved in any coherence transaction. The limited directory scheme can be extended to allow the software to emulate a full-map directory protocol [CKA91, ACJ+99].

A *chained directory protocol* uses a linked list to track all of the sharers [Gus92, JLGS90]. Each sharer points to the next node that has a cached copy of the data, with the directory entry keeping the head of the list. Such an approach does not incur software emulation overhead, but has poor invalidation latency because the entire linked list is traversed linearly.

The *coarse vectors* approach [AGGD01, LL97, MH94] uses each bit in the presence vector to point to a set of nodes instead of a single node, even though not all nodes in the set are necessarily sharers. Compared to the full-map approach, coarse vectors have less precision and can generate false sharing traffic, but can have much smaller directory size. Readers can find extensive summaries of various cache coherence schemes in [Ste90, CSG97, SBD+97].

2.4 Latency Reduction Techniques for DSM

The performance of DSM systems depends heavily on the memory access latency of the underlying hardware. In this section, we study various latency reduction techniques previously proposed for DSMs, including *prefetching*, *multi-threading*, *remote caching*, and *cache-only memory architectures (COMA)*.

2.4.1 Prefetching

Prefetching is a mechanism that loads data into the cache or local memory before it is actually used, anticipating that it will be used in the near future [CKP91, MG91, BC91, Lee87]. When applied to a DSM, proper prefetching could avoid the long stalls created by fetching data from far-away memory modules. Software prefetching mechanisms [CKP91, MG91] are directed by the compiler, using static analysis to strategically embed explicit non-blocking prefetch instructions in the code sequence. To be effective, the prefetch instructions must precede far ahead of the data fetch, improving the chance that the data will be in the cache when it is needed. Simple hardware prefetching [Smi82] sequentially fetches the next cache block according to address. More sophisticated hardware mechanisms try to detect simple address patterns, such as constant strides, and prefetch accordingly [BC91, Lee87]. Since hardware prefetching guesses which data will be used, increased remote traffic could become a concern.

2.4.2 Multi-threading

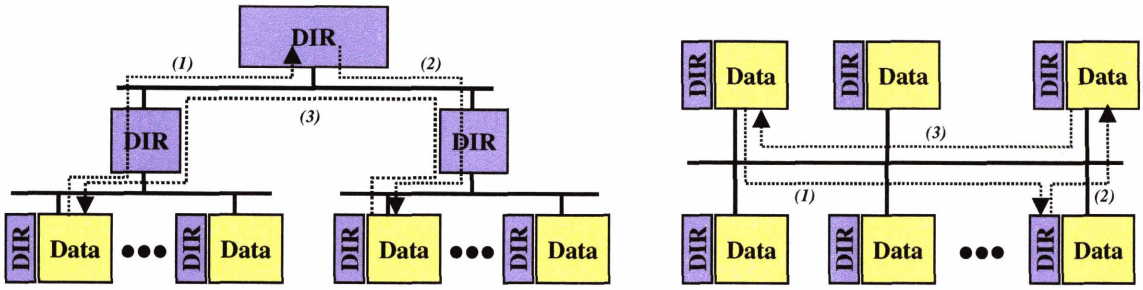
Multi-threading [ALKK90, LGH94] hides long memory access latency by switching among multiple hardware threads active on each processor. Its success hinges on two important factors. First, the underlying hardware must support low-overhead multi-threading capabilities with a fast context switch. Second, the workload itself must have favorable data access patterns among the threads sharing the same cache, so that the context switch does not thrash the cache content on each node. If there are enough threads waiting, multi-threading can hide the latency well and yield high throughput. However, multi-threading cannot reduce the latency each individual thread experiences, and cannot reduce remote traffic.

2.4.3 NUMA with Remote Cache

NUMA with Remote Cache (*NUMA-RC*) [ZT97] uses a large block of DRAM at each node to form a *local remote cache*. Upon a cache miss to a cache block located in a remote memory module, the block is brought into both the regular cache and the remote cache of the requestor. Therefore, the local remote cache is likely to hold the working set of the local thread over time. All of the blocks in the remote caches are kept coherent by the main memory directory.

2.4.4 Cache-Only Memory Architectures

Similar to NUMA-RC, *cache-only memory architectures* (*COMA*) also use local memory to hold the working set for the local thread. The main difference between COMA and NUMA-RC is that in COMA, a data block is not stored at the home node, but rather resides on the nodes where it is used most often. The local memory is referred to as *attraction memory*



(a) Hierarchical COMA *(b) Flat COMA*

Figure 2-5: Illustration of hierarchical and flat COMAs.

because the data block is brought (attracted) into the cache and the local memory of the requestor. Because a data block in a COMA machine can reside on any node, the process of locating any given block becomes more complex compared to NUMA-RC. Next, we describe several different COMA designs that use different localization schemes.

Hierarchical COMA

One of the earliest COMA machines is the data diffusion machine (DDM) introduced in [HLH92] (Figure 2-5(a)). The DDM uses a tree-like hierarchical approach to locate a data block. At the root of each subtree, a directory records all the data stored in that subtree, with the actual nodes and data as the leaves of the tree. Therefore, in order to locate a block, a traversal of the tree suffices. The requesting nodes initiates the lookup, traversing upward toward the root of the tree. The upward traversal stops when the request reaches the subtree root that contains the directory information of the requested block. The directory information is used to obtain the data within the subtree. Figure 2-5(a) shows an example in which the information of the requested block is held at the root of tree.

Flat COMA

Since the process of locating a cache block in a hierarchical COMA system is rather complex, a *Flat-COMA* architecture [SJG] simplifies this process by storing the location information of a cache block at its home node, which is statically determined by its address (Figure 2-5(b)). Each memory access locates the block by consulting the home node as shown in Figure 2-5(b).

Simple COMA

Simple COMA [SWCL95] partitions the task of data management into a software component and a hardware component. Simple COMAs use the operating system to manage the data allocation in the attraction memory, and use hardware to manage data coherence.

Because data migration is done in software, sophisticated algorithms using software hints can be used to better direct the data movement around the system. However, the operating system must move data on a page granularity, thus managing coherence in software would cause significant overhead because each miss would trigger a page fault. Therefore, data coherence is left to hardware and performed at a cache block granularity. One concern often encountered in simple COMA, however, is that spatial locality at page granularity is low, thus significantly under-utilizing the memory space.

2.4.5 Summary

In this chapter, we reviewed some basics of multi-chip multiprocessor systems. We drew parallels between DSM and CMP systems, especially between the NUMA and NUCA properties. In particular, we presented several well-known techniques for memory fetch latency reductions for NUMA machines. While NUMA and NUCA present similar problems, these latency reduction techniques cannot be directly applied to CMPs. Specifically, in CC-NUMAs, the allocation of the local cache between private and shared data only affects the local node performance because they are private to the node. Furthermore, in CC-NUMAs and COMAs, remote data is further away than local DRAM, thus it is beneficial to use local DRAM as remote caches, which is both cheap and does not reduce the local L2 cache performance.

Chapter 3

Memory Hierarchy Architecture and Implementation

In this chapter, we describe the implementation of the baseline private and shared cache designs introduced in Chapter 1. The designs are instantiated on a specific underlying CMP organization which we refer to as a *Tiled CMP*.

3.1 Tiled Single-Chip Multiprocessors

As more and more cores are placed on future CMPs, the bandwidth and latency of the interconnection network in the "dance-hall" style CMPs will become a bottleneck. We believe that in the future, on-chip interconnect will move away from a shared bus to a switched network. CMP designs will naturally evolve toward arrays of replicated tiles connected over these networks to further reduce the re-design effort of the communication network. These tiled CMPs scale well to larger processor counts and can easily support families of products with a varying number of tiles.

In this thesis, we focus on a class of tiled CMPs where each tile contains a processor with L1 caches, a slice of the L2 cache, and a connection to the on-chip network, as shown in Figure 3-1. This structure closely resembles a shrunken version of a conventional mesh-connected multi-chip system. To maintain cache coherence, we use a directory-based coherence protocol to facilitate scaling to larger node counts. The rest of this chapter uses the tiled CMP as the baseline design to describe how the private and shared designs are implemented.

3.2 Basic Assumptions

This section details some basic design assumptions applied to all designs in this thesis. We assume all CMPs are based on a unit tile replicated in a 2-D mesh configuration, as shown

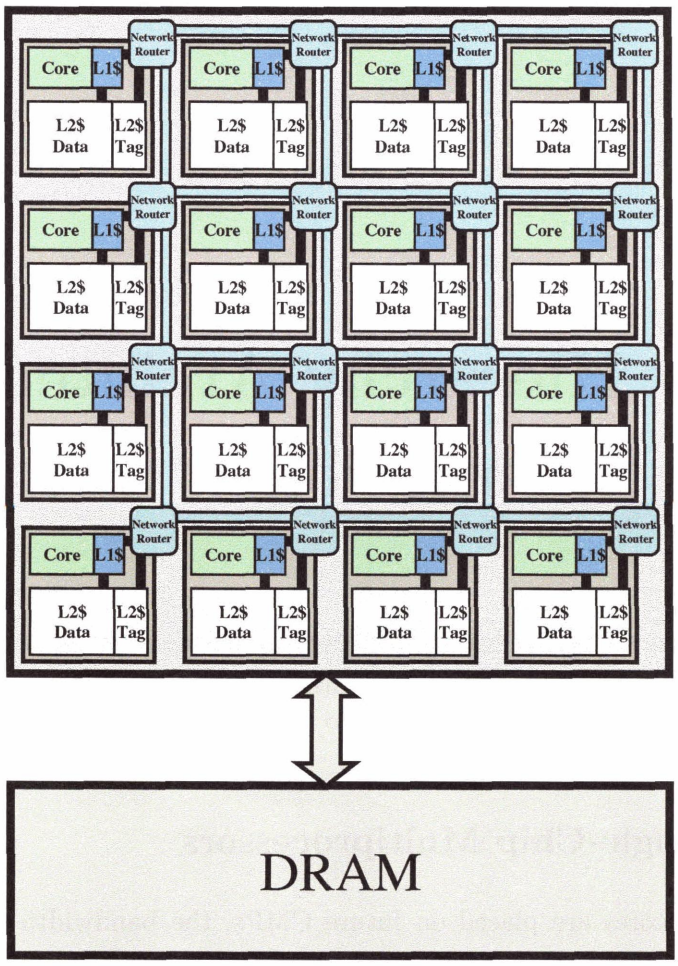


Figure 3-1: Tiled CMPs are a subset of CMPs where each tile contains a processor with L1 caches, a slice of the L2 cache, and a connection to the on-chip network. This structure resembles shrunken versions of a conventional *mesh-connected* multi-chip multiprocessor system. A 2D mesh routing network is used to connect all the tiles in the system. Cache coherence is maintained through a scalable directory-based protocol.

in Figure 3-1. Each tile contains a processor core, separate L1 instruction and data caches, a unified L2 cache storage with any associated directory information, and a network switch. Additional assumptions are as follows:

1. The L1 instruction and data caches are not the focus of this thesis. They are private to the processor core and are kept small compared to the L2 caches. To provide the lowest possible latencies, L1 caches are tightly integrated with the processor.

2. The local L2 storage is tightly coupled to the rest of the tile and is accessed with a fixed latency pipeline. The tag, status, and directory information are kept separate from the data arrays and close to the processor core and network router for quick tag resolution.

3. All the caches in our system are non-blocking. A miss buffer is used to store current

misses, allowing future requests for different addresses to proceed. Figure 3-2 shows the data access path in our baseline system.

4. Access to L2 slices on remote tiles travels over the on-chip network and experiences varying access latencies, depending on the inter-tile distance and network congestion.

5. The on-chip interconnection network used in this thesis is a deterministic wormhole routed virtual channel network arranged in a 2D mesh. Figure 3-3 shows the router architecture, which has two physical links per direction (one input channel and one output channel). Each physical input channel has two virtual channels to avoid deadlock.

6. To improve scalability, a directory-based protocol is used as the basis for all the coherence schemes discussed in this thesis. Each directory entry uses a rudimentary full-map (one bit per tile) presence vector to keep track of the sharers.

7. A request-reply, invalidate-based, four-state MESI protocol, with reply-forwarding, is used as the baseline cache coherence protocol, with each design using a minor variant. More detailed discussion about the protocol features and implementations will be presented in Section 3.5 and Appendix A.

3.3 Private Design

In the private design shown in Figure 3-4, the processor core uses the local L2 slice as a private L2 cache. This approach is used by several commercial CMPs, such as the Intel Montecito [CR05] and AMD's Opteron [KMAC03].

The operation of the private design is straightforward. When an L1 miss occurs, it is forwarded to the local private L2 cache, and a hit in the private L2 cache completes the fetch. The miss scenario is more complicated because the directory entry must be consulted to maintain data coherence for all of the L2 copies of the requestor data block. Because each memory block is associated with a directory entry, the directory area overhead of using a full-map directory can be significant as discussed in Section 2.3.2. Therefore, most directories are kept in off-chip memory because the area necessary to place them on-chip is unrealistic.

The main issue in using an off-chip directory is that its access latency is much higher compared to on-chip communication latencies. This problem has not been severe in multi-chip multiprocessor systems because most of the time the requested data is also in off-chip DRAM modules, which must incur a long fetch latency anyway. For a CMP, however, the difference between on-chip and off-chip latencies is dramatic.

In a naive implementation of the private design, even if a shared data block is present in the private L2 cache of another tile, the L2 miss is not aware of their presence until it

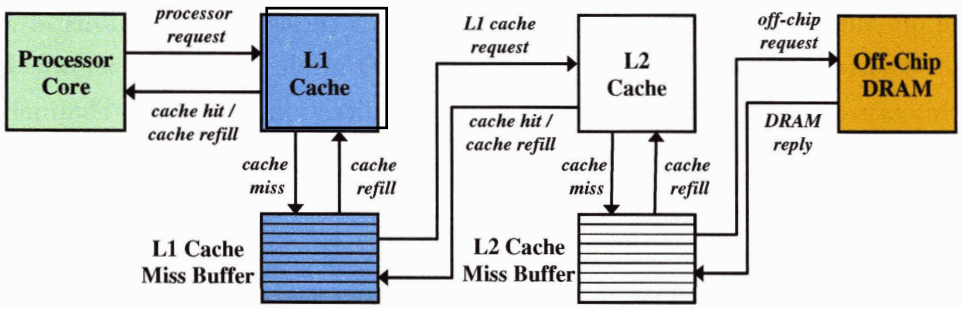


Figure 3-2: The access path of the non-blocking two-level cache hierarchy used in this thesis. Each cache miss, writeback request, or explicit drop request is kept in a miss buffer to allow future accesses to proceed. Misses to the same address are merged into a single entry in the miss buffer when appropriate. Future misses to different addresses are not blocked as long as there is an available entry in the miss buffer.

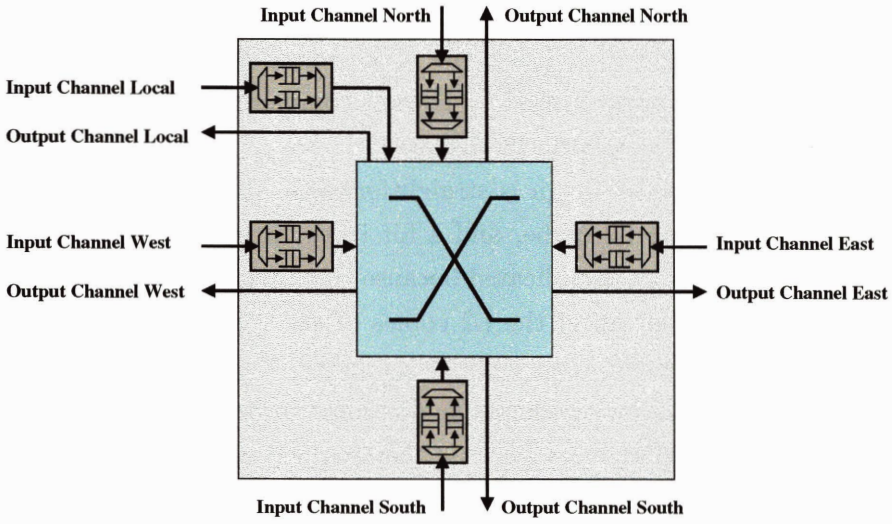


Figure 3-3: A two-dimensional mesh router with two physical channels per direction and two virtual channels per physical channel.

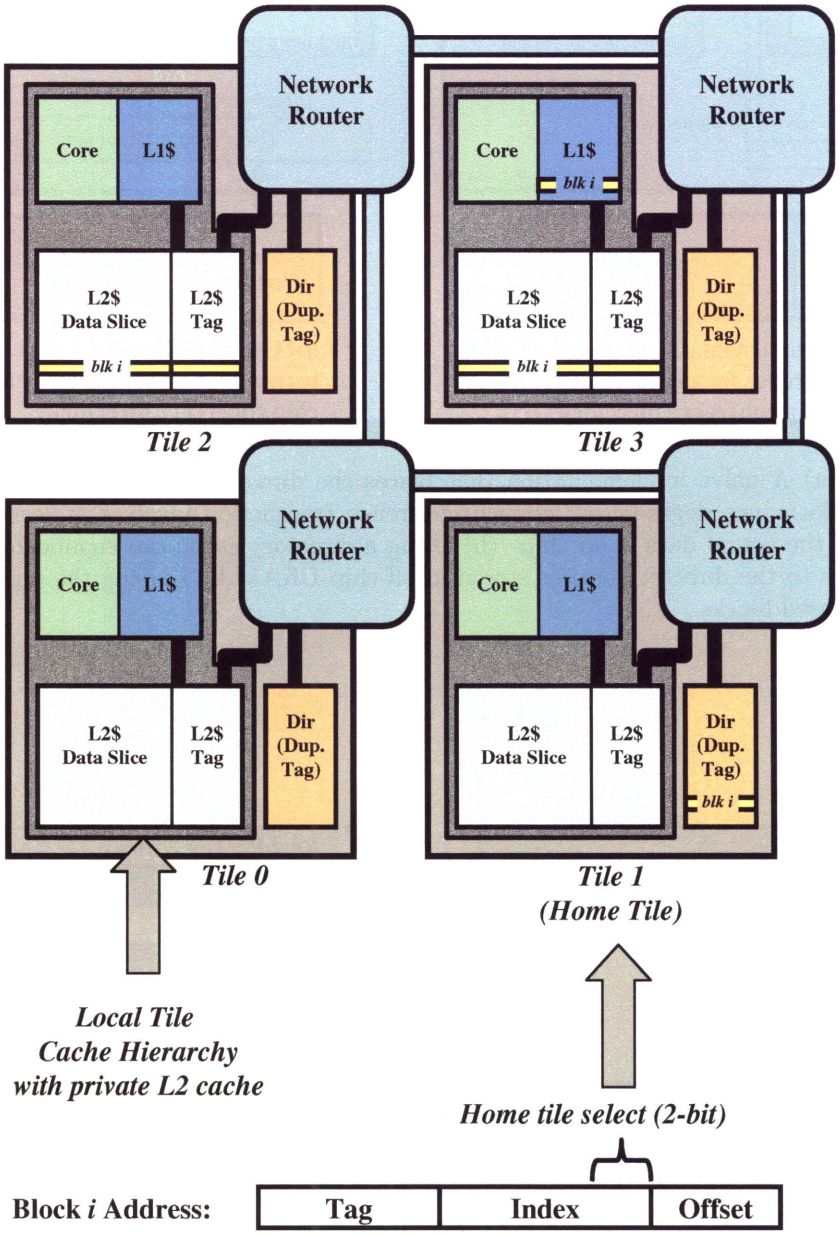
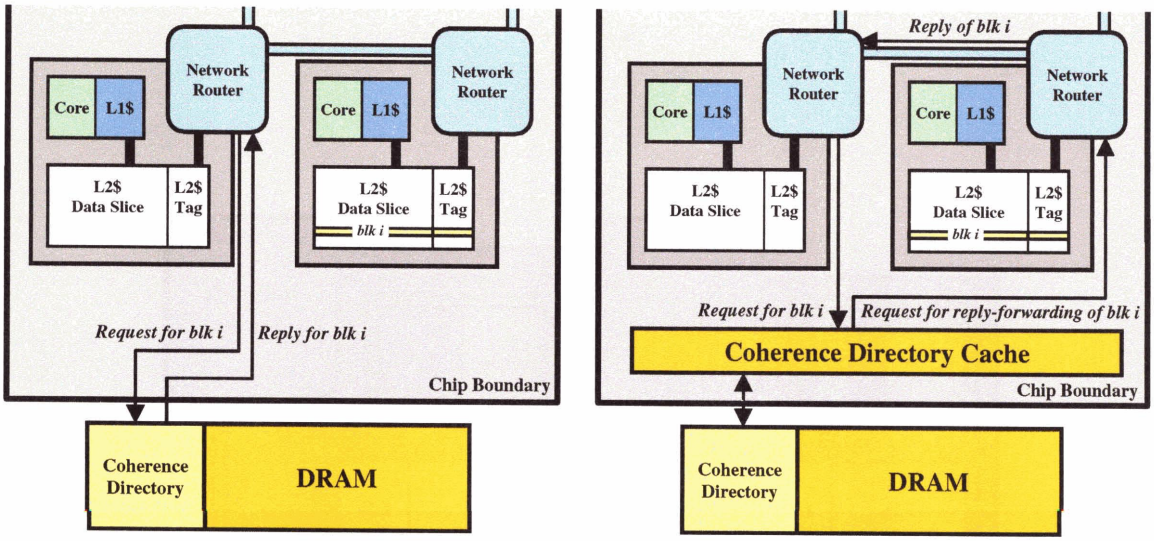


Figure 3-4: In a private design, each processor core treats its local L2 slice as a private L2 cache. Shared data must be copied to the private L2 caches of all the sharers. Thus, data coherence must be maintained among all L2 caches.



(a) Naïve implementation of directory-based coherence for private design with off-chip coherence directory

(b) On-chip directory cache is used to avoid long access latency of off-chip coherence directory

Figure 3-5: (a) A naive implementation that places the directory in off-chip DRAM can suffer significant performance degradation as each coherence transaction involves at least one off-chip access, even if the actual data is on chip. (b) Using a directory cache can significantly reduce the access latencies to the directory entries stored in off-chip DRAM by keeping the directories of the most recently used blocks.

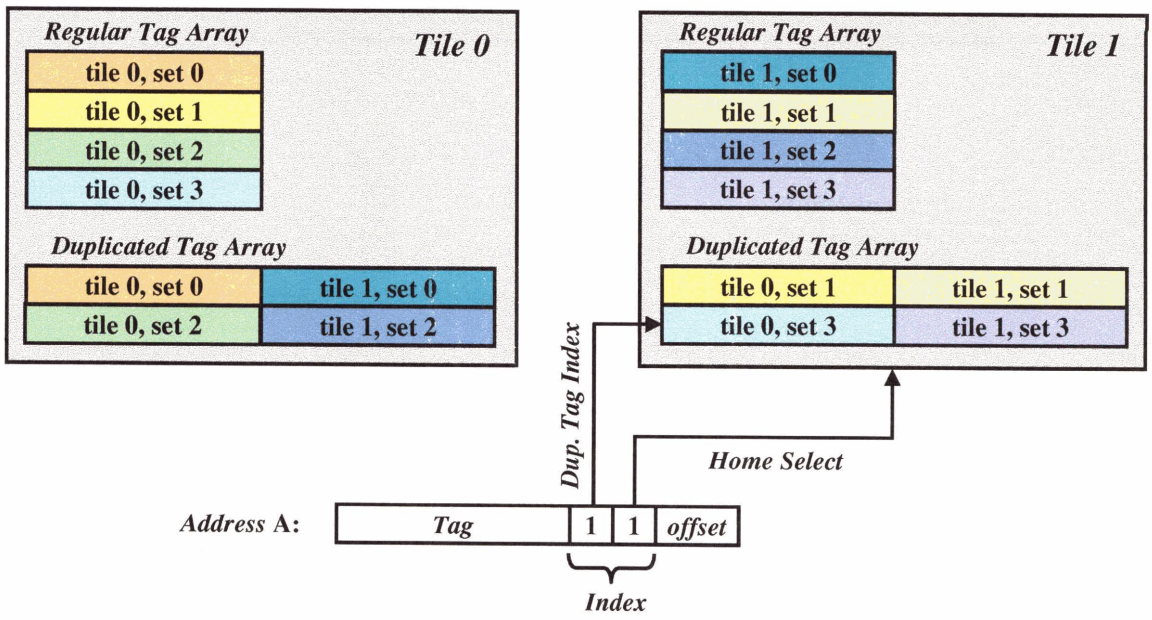


Figure 3-6: Example of using duplicated L2 cache tags to implement an cache coherence directory. Each L2 tag is duplicated and stored at its home node, determined statically by address. Directory information is deduced from the collection of the L2 tags.

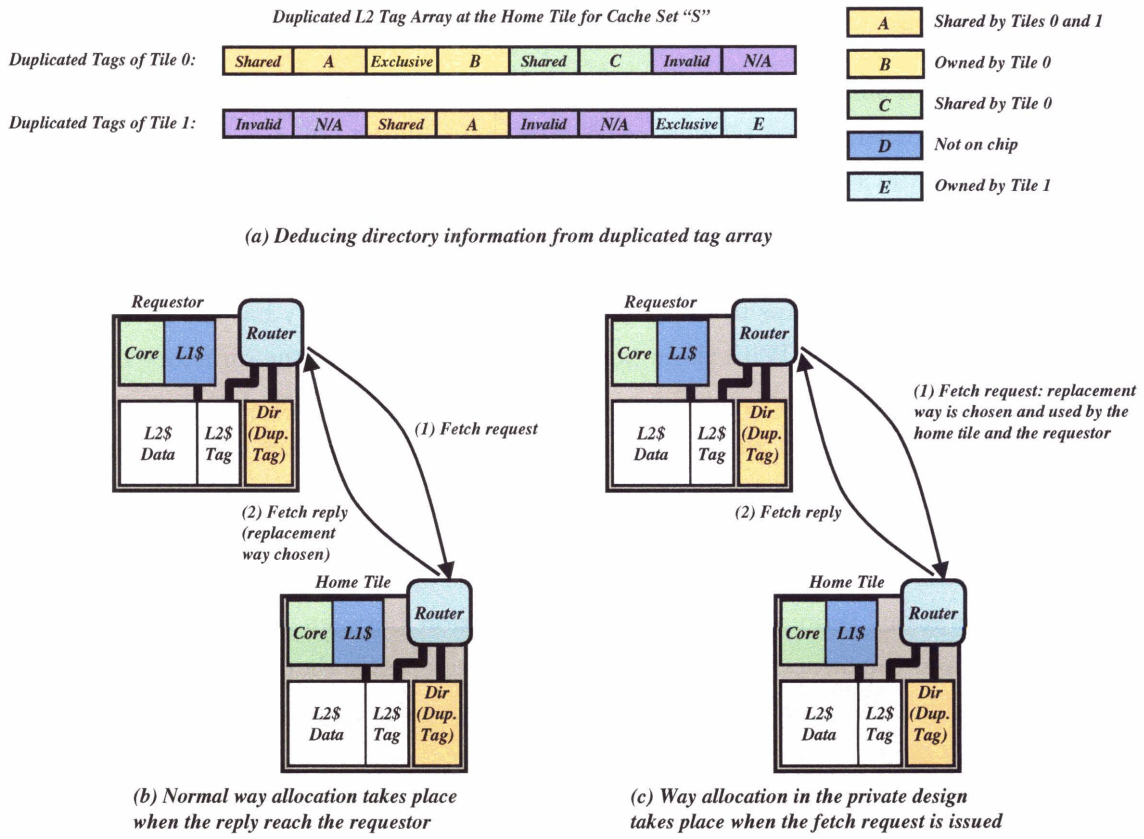


Figure 3-7: Examples of the duplicated-tag directory for the private design.

accesses the off-chip memory, e.g., *block i* shown in Figure 3-5(a). Using an on-chip *directory cache* is one approach to reduce off-chip directory lookups by keeping an small subset of the entries on-chip, as shown in Figure 3-5(b). Directory caches are simple to implement and can be very effective, depending on the data access patterns of the cache. However, in our simulations, using a directory cache did not lead to a high enough hit rate for our benchmark suite. Thus, we opted to implement a duplicated-tag directory scheme, which can be placed on-chip with a moderate area requirement.

3.3.1 Duplicated-Tag Directory Implementation

The goal of the duplicated-tag directory is to keep the directory entries of all the cached L2 blocks on-chip. The directory entries are held as a duplicate set of L2 tags distributed across tiles by address [BGM+00]. For each processor accessing a particular cache block, a copy of the block must be resident in its private L2 cache, such as *block i* shown in Figure 3-4. In addition, an on-chip directory holding an entry for *block i* is stored at *block i*'s home tile, statically determined by the *home select* bits of the address, which in our case, is the lower bits of the cache index form the home select.

Directory Usage

Figure 3-6 shows a simplified two-tile example of how this scheme works. In this example, each tile has a direct-mapped L2 cache with four cache blocks. We use the *home select* bits to find A's home tile and determine A's status on-chip. The remaining bits of the index are used to find the duplicated tag entry corresponding to A in the directory on the home tile. This entry stores the duplicates of all L2 tags in the cache set that A maps to from all the tiles. In this example, A maps to set 3, and the duplicated tag entry has the L2 tags of set 3 from both tile 0 and tile 1. With these tags, we can easily deduce the directory information of A. Therefore, we have constructed a perfect directory for all of the data currently cached on chip. Figure 3-7(a) shows an example of a two-tile system and how the state and sharing information of data blocks A to E can be deduced.

The main drawback of this approach is the area overhead, which we will discuss in Chapter 6. Cache-to-cache transfers are used to reduce off-chip requests for local L2 misses, but these operations require three-way communication between the requesting tile, the directory tile, and the owner tile. This operation is more costly than hits to global locations in a shared design, where a three-way cache-to-cache transfer only occurs if the block is held in the exclusive state.

Directory Maintenance

One complexity in maintaining the duplicated-tag directory is that the tags in the directory must be identical to the actual L2 tags they shadow, or the directory would encode the wrong sharing information. Therefore, each time the tag changes, the directory must also be updated.

Normally, way allocation and any necessary writebacks in each cache set are done when the requested data reaches the requesting tile, as shown in Figure 3-7(b). In the private design, however, the L2 cache of the requestor and the directory on the home tile must agree to use the same way for each refill data block. Therefore, we choose to select a way to refill into (and necessary writebacks) and send that information to the home tile at request time. When the requested data is returned, the directory has already updated the tag information using the replacement way agreed upon to reflect the most up-to-date sharing information.

3.4 Shared Design

In the shared design, all of the L2 slices are managed as a single shared L2 cache with addresses interleaved across slices. The shared design is used by a number of commercial CMPs, such as IBM's Power series [TDJ+02], Sun Microsystems' Niagara [Kre04b], and Raza Electronics's XLR series [Raz05].

Figure 3-8 shows the implementation in detail for our tiled CMP. On-chip L2 storage is split evenly among all tiles but logically forms one large cache. On an L1 cache miss, the

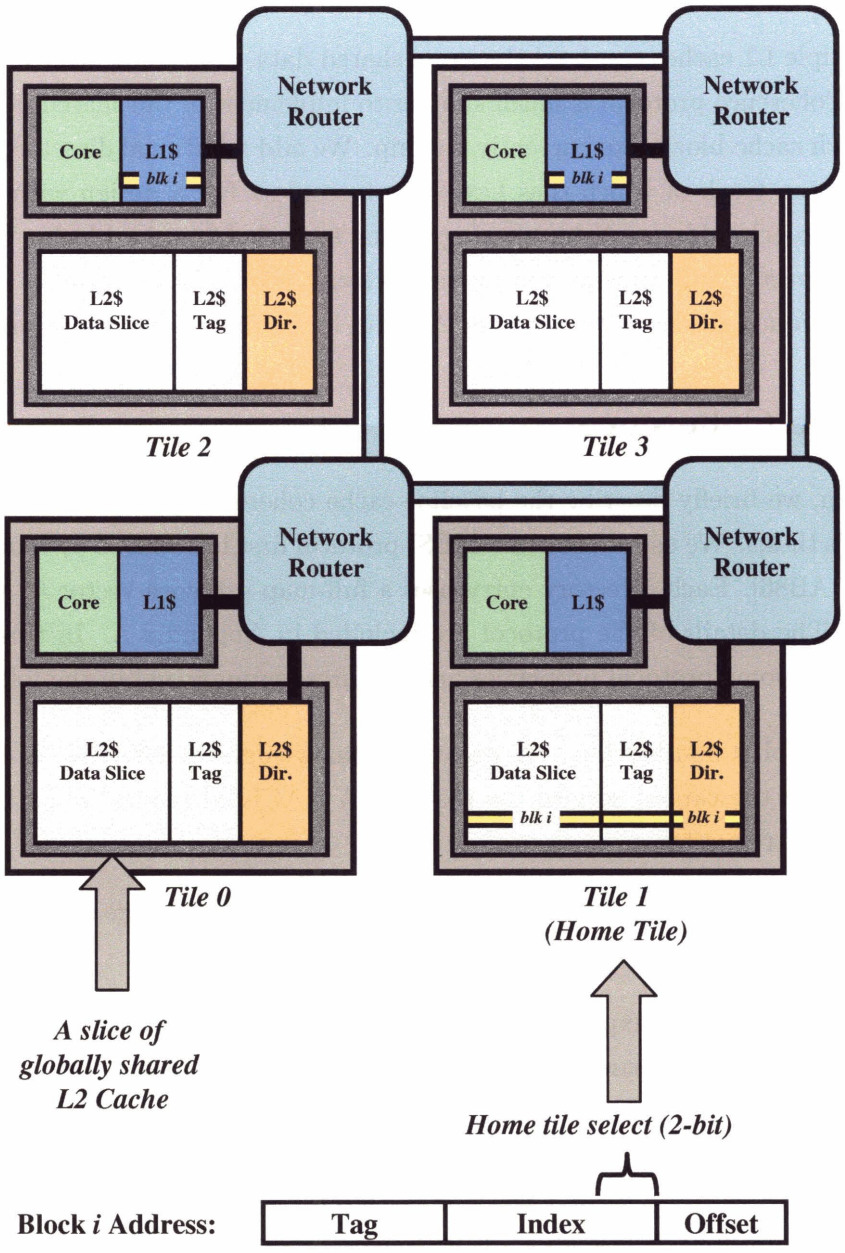


Figure 3-8: In a shared L2 design, all of the on-chip L2 slices are aggregated to form a single large logical L2 cache. Each L1 cache miss must travel to the home node of requested block to access the data. Data coherence is maintained for all the L1 sharers.

fetch request is forwarded to the requested block's home tile, which could be either local or remote. Latency to the L2 slice varies according to network congestion and the number of network hops between the requesting processor and the home tile.

On-chip Directory Implementation

Because multiple L1 caches can hold the same shared data, coherence must be kept among all the L1s. Coherence protocol is much simpler to implement in the shared design because we know which cache blocks are currently on-chip. We add additional directory bits to each L2 block, to keep track of which tiles have remote copies. For a design with N processor cores, this approach adds an N-bit sharing vector to each L2 cache block. The overhead of the sharing vector will grow as the processor count grows, but a number of previously proposed techniques, discussed in Chapter 2, could be used to reduce directory overhead.

3.5 Cache Coherence

In this section, we briefly describe the baseline cache coherence protocol used for all cache designs in this thesis. We use a four-state MESI protocol first introduced by Paramarcos and Patel [PP86, AB86]. Each directory entry uses a full-map presence vector to store sharing information. The details of the protocol are included in Appendix A. In this section, we simply highlight some protocol properties and features, summarized in the following:

1. The protocol is non-blocking. A negative acknowledgment (*NACK*) is used as reply if the home tile cannot service the request. A NACK'ed request must be retried by the original requestor.

2. The protocol does not assume any network ordering of its message delivery. Coherence messages can be reordered or delayed arbitrarily.

3. The protocol requires explicit drops of all clean cache blocks, i.e., the directory must be informed when a clean cache block is evicted from the local cache.

4. The protocol dynamically backs off requests in a race condition to avoid starvation of any of the requestors.

5. The protocol acknowledges all explicit drops and writeback requests.

3.6 Summary

Private Design Recap

The private design has low L2 hit latency, as the L2 is physically co-located with the processor core and has much smaller area than a shared cache. This design provides good

performance when the working set fits within the local L2 slice. Its main disadvantage is that effective on-chip cache capacity is reduced for shared data because each core must retain its own copy of any shared data block. Furthermore, the fixed partitioning of resources does not allow a thread with a larger working set to "borrow" L2 capacity from the private caches of other processors hosting threads with smaller working sets.

Shared Design Recap

The shared design minimizes the off-chip miss rate for large shared working sets, as only a single on-chip copy is required for any shared data. However, two significant drawbacks may reduce the effectiveness of the shared design. First, large shared L2s will have worse access latency than a small private L2 even when each physical L2 slice is optimally sized for access latency. This is due to the increasing global wire delay that makes transferring data across chip expensive. Second, the associativity of the L2 cache needs to be high enough to accommodate the number of on-chip threads. Otherwise, we may suffer from inter-thread cache conflicts, especially for applications that have little sharing. As more tiles are anticipated in future systems, the off-chip misses caused by the inter-thread conflicts may outweigh the savings from increased capacity.

Chapter 4

CMP Latency Reduction Techniques

In Chapter 3, we introduced two baseline L2 cache designs. First, a *private design* dedicates a slice of the on-chip L2 cache storage as a private L2 cache for each processor core. Second, a *shared design* aggregates all the on-chip L2 cache capacity to form a single L2 cache shared by all the processor cores. These two designs illustrate the trade-offs between two key components that control effective memory access latency, namely, *on-chip access latency* and *off-chip miss rate*. Figure 4-1 shows this trade-off.

4.1 Hybrid Designs

Each workload has specific characteristics that could lead to considerably better performance with either a private or a shared design. Furthermore, each workload itself may be divided into several distinct program phases that call for different designs. This intuition has been shown by many recent studies [HKS+05, CPV05]. In Chapter 6, our results also confirm this hypothesis.

This observation is the chief motivation to develop hybrid cache system architectures that retain the advantages of both private and shared designs. The main design goal of any hybrid design is to achieve lower off-chip miss rate than the private design and lower on-chip access latency than the shared design, as shown in Figure 4-1.

In this thesis, we present two hybrid designs, *victim replication (VR)* and *victim migration (VM)* that try to reduce both on-chip access latencies and off-chip miss rates to yield better performance than either private or shared design. Both victim replication and victim migration are based on the shared design. We show that by using victim replication, we can trade a small increase in the off-chip miss rate for significantly reduced on-chip fetch latency. Victim migration has a slightly higher area overhead than victim replication but is more flexible and can fully mimic the behavior of the private design, and is particularly

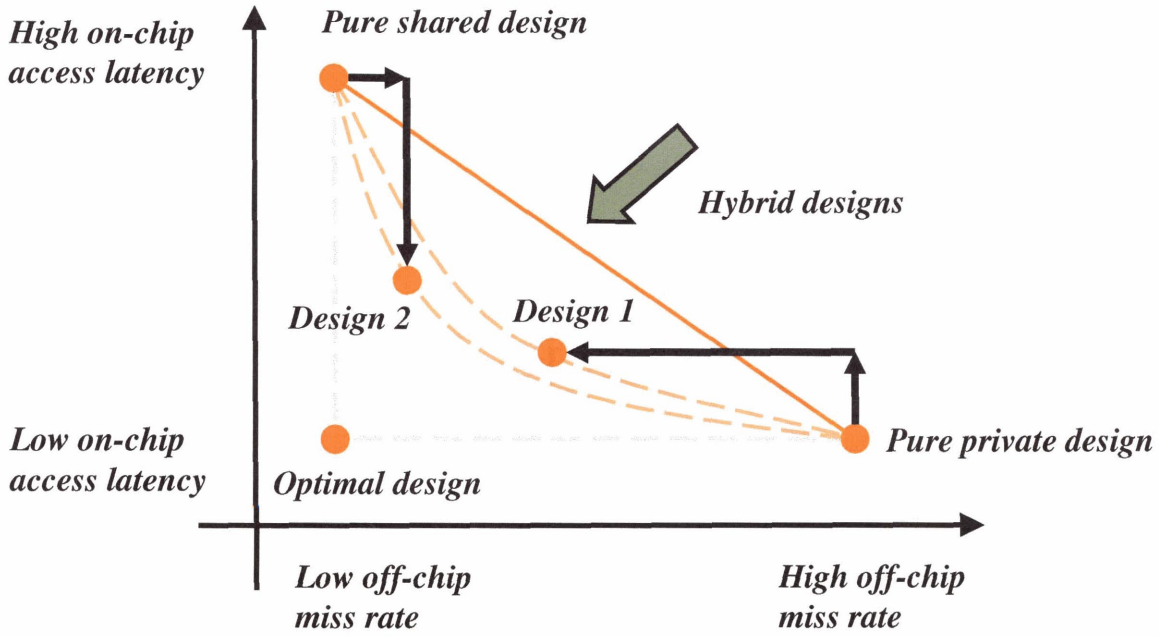


Figure 4-1: The trade-offs between two conflicting goals in designing a hybrid on-chip cache architecture: *off-chip miss rate* and *on-chip fetch latency*.

well-suited for multi-programmed workloads.

4.2 Overall Design Approach

Before explaining our hybrid designs in detail, we first discuss how we approach designing a hybrid layout in a CMP cache system that combines the advantage of the private and the shared designs.

4.2.1 Improving the Bottomlines

Figure 4-2 shows a generic four node CMP, with each node having a slice of the L2 cache space. Depending on whether we use the private or the shared design, each cache block falls into one of three categories: (1) an *unshared (private) block*, where the host node is the only user of this cache block; (2) a *global shared block* in its statically mapped home location; and (3) a *replicated shared block*, where each sharer replicates a copy in its local cache slice.

With a pure private design, a cache slice can contain either private blocks or replicated shared blocks, but not global shared blocks. However, introducing shared global blocks into a pure private design can be beneficial. First, if the capacity in a particular cache slice is not fully utilized, the unused space can store shared global blocks for other nodes, creating a limited form of cache capacity stealing to reduce the off-chip miss rate. Second, if the working set does not fit into the local cache slice, we can increase the effective on-chip cache

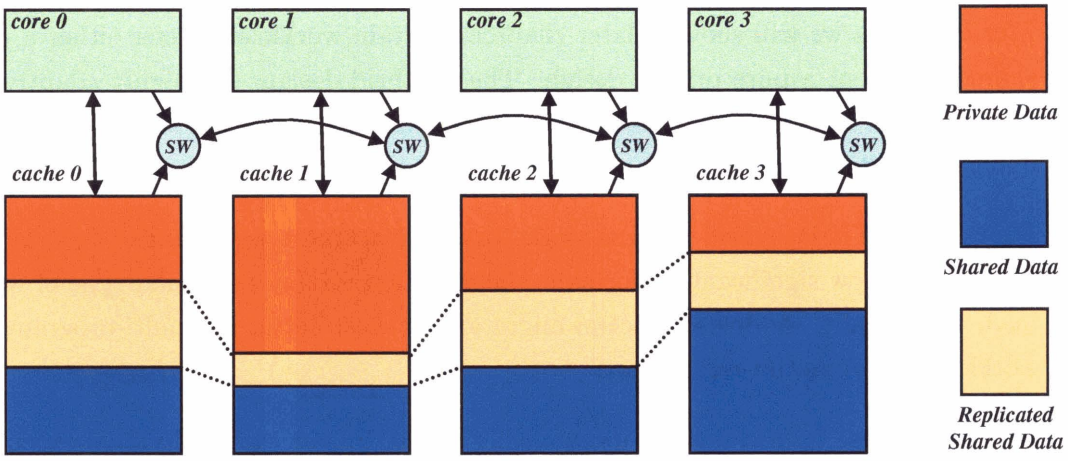


Figure 4-2: Illustration of the hybrid design approach. Three different types of blocks can be present in a hybrid design: private blocks, global shared blocks, and replicated shared blocks.

capacity by replacing some replicated shared blocks with global shared blocks. While fewer replicated shared blocks can lead to more cross-chip fetches, the increased on-chip capacity can reduce costly off-chip misses, creating an overall performance gain.

With a pure shared design, each cache slice contains only global shared blocks. Allowing replicated shared blocks in a pure shared design can also be helpful. First, if the capacity of a slice is not saturated, the unused space could store replicated cache blocks local to that node, turning some long cross-chip fetches into local ones. Second, if an often accessed block is in a distant location from its requestor, allowing the requestor to having a local copy of the block could significantly improve overall fetch latency, even if this means that another global shared block must be evicted in order to accommodate this replicated block.

Therefore, a hybrid design allowing all three types of blocks to co-exist can potentially perform better than both the private and the shared designs. In creating such a hybrid design, we must first craft a mechanism that allows the caches to be divided into two partitions, a *shared partition* holding global shared blocks, and a *private partition* holding private and replicated shared blocks. Moreover, we must also devise management policies to solve two problems, namely, how to determine what is the right division between the two partitions, and what data to place in which partition. Victim replication and victim migration use similar partition mechanisms, and each details a set of management policies that achieve superior cache performance than both the private and the shared designs.

4.2.2 Design Criteria

Besides achieving good performance, our hybrid designs also have several other highly desirable properties. They are summarized in the following:

1. *Simplicity*: These designs do not introduce significant additional complexity or overhead to the baseline system.

2. *Flexibility*: As we will show in later chapters, certain workloads prefer either a pure shared design or a pure private design. These hybrid designs are highly adaptive to closely mimic the behavior of these two baseline designs and avoid significant performance degradation from each baseline.

3. *Robustness*: These hybrid designs work very well across a wide range of workloads and do not show significant performance degradation for any particular type of workload. Specifically, we devised victim migration to work better for multi-programmed workloads than victim replication.

4. *On-Line*: These hybrid designs dynamically adjust to suit each individual execution phase within each benchmark. Several proposed static designs use profiling information to determine the best suited hybrid design. However, many workloads display clear execution phases that may call for different designs during the execution.

4.3 Victim Replication

Victim replication is a simple hybrid approach based on the shared design. Its main idea is to use the local L2 cache slice to capture some of the evictions from the local L1 cache. Each retained victim is a local L2 *replica* of a block that already exists in the L2 cache at the remote home tile. This idea is shown in Figure 4-3. A significant number of future accesses will hit in the capacity victim replicas, thus providing short fetch latency by efficiently creating a local victim cache in the L2 slice.

4.3.1 Mechanisms

When a processor request misses in the shared L2 cache, a cache block is brought in from memory and placed in the on-chip L2 at its home tile, just as in the shared design. The requested block is also forwarded to the L1 cache of the requesting processor. If the block's residency in the L1 cache is terminated because of an incoming invalidation request, we simply follow the usual protocol of the shared design and invalidate the L1 cache copy. If an L1 cache block is evicted because of a conflict or capacity miss, we *attempt* to keep a copy of the victim block in the local L2 slice to reduce subsequent access latency to the same block. In some instances, we may choose not to replicate the victim, as described below.

All primary cache misses must now first check the local L2 tag array in case there is a valid local replica. On a replica miss, the request is forwarded to the home tile following standard protocol. On a replica hit, the replica is invalidated in the local L2 slice and moved into the L1 cache, completing the request. When a downgrade or invalidation request is received from the home tile, the L2 tag array must also be checked in addition to the L1 cache tag array to maintain coherence.

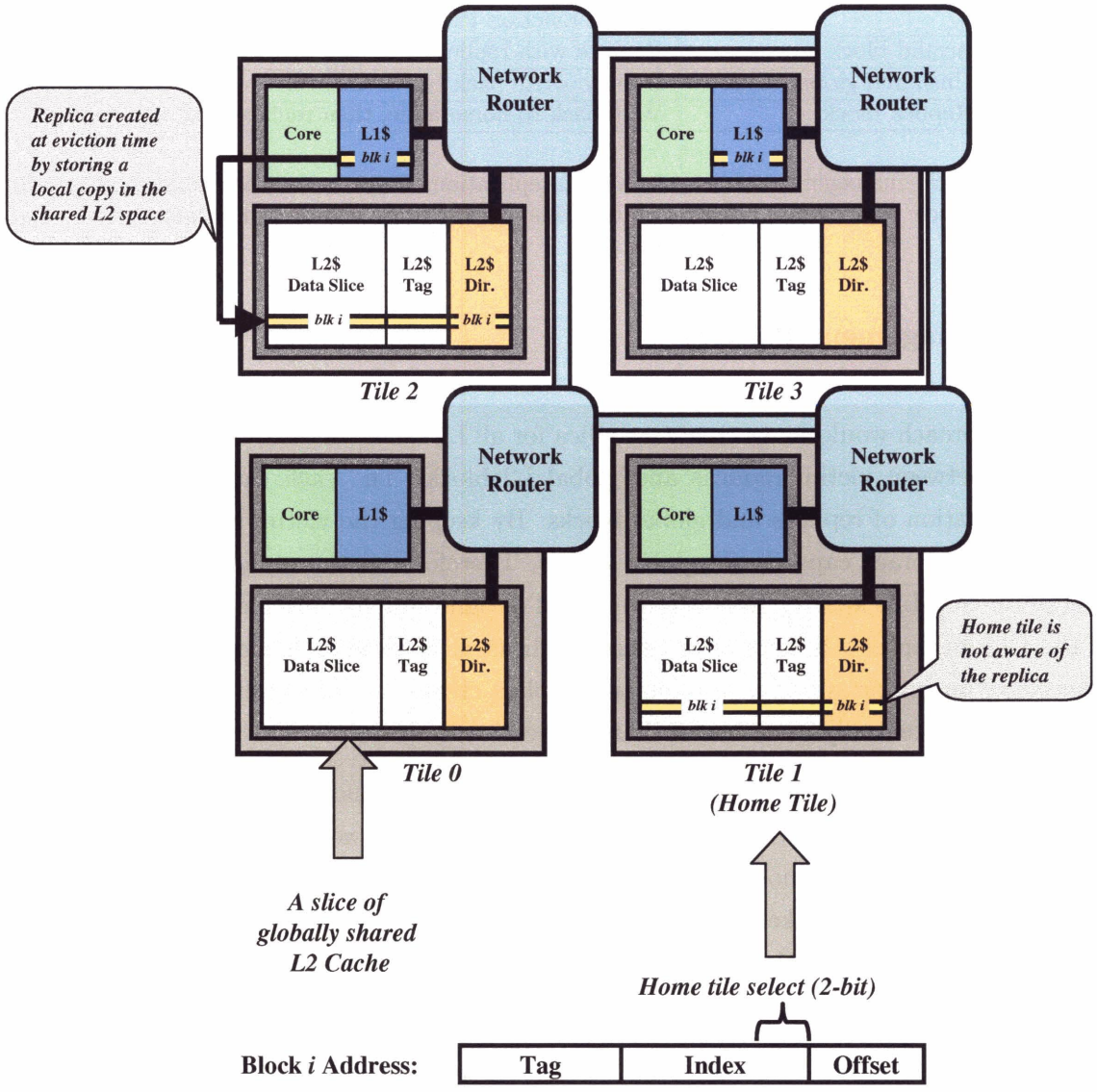


Figure 4-3: *Victim replication* is a simple hybrid design that combines the large capacity of the shared design with the low hit latency of private design. Victim replication is based on the shared design, but in addition tries to capture evictions from the local L1 cache in the local L2 slice, such as the L2 copy of block i captured by Tile 2. Each retained victim is a local L2 *replica* of a block that already exists in the L2 of the remote home tile.

Priority	Target Block Type	Action
L2 Cache Refill Policy		
1	Invalid block	Refill
2	Unshared global block	If dirty, write back to DRAM, then refill
	Replica block	Writeback to home node, then refill
3	Shared global block	Invalidate all sharers, write back if dirty, then refill
L1 Cache Eviction Policy		
1	Invalid block	Replace with replica
2	Unshared global block	If dirty, write back to DRAM, then replace with replica
	Replica block	Writeback to home node, then replace with new replica

Table 4.1: Cache management policies for victim replication. Blocks are chosen in descending order according to their priority and blocks with the same priorities are chosen at random.

4.3.2 Management Policies

A naive approach would be to create a replica for all L1 cache victims, but L2 slice capacity is shared between victim replicas and global L2 blocks, i.e., each cache set can contain any combination of replicas and global blocks. By keeping the victim replicas, we are also reducing the storage capacity for global blocks. Therefore, victim replication will have less overall on-chip L2 capacity than a pure shared design. But by creating replicas, a fraction of the L2 hits can now be serviced by these replicas, thus avoiding longer cross-chip fetches. Therefore, an important task in managing replica creation is to *not* evict a global shared block if it is potentially more useful than the replica itself.

We choose to use the sharing information of a block to evaluate its current usefulness. If a global shared cache block is currently shared by another node, we deem it useful. Conversely, if a global shared block is not used by anyone, i.e., has no sharers, it is considered less useful and can be evicted to make room for a replica. This observation forms the basis for both victim replication and victim migration.

In the following, we detail our heuristics to efficiently manage the on-chip cache capacity. Specifically, we discuss two policies to manage way replacement in a cache set. First, the *L2 refill policy* determines where to place a cache block when the L2 receives a reply for an L2 miss from off-chip memory. Second, the *L1 eviction policy* determines whether to replicate, and if so, where to keep an L1 victim in the local L2 slice. Table 4.1 summarizes the policies.

With victim replication, there can be four types of cache block that live in a cache set: (1) an invalid block; (2) a replica block; (3) a global block that currently has L1 sharers; and (4) a global block that currently does not have any L1 sharers. The management policies describe the process used to choose from these four types of blocks when looking for a space to store either an off-chip memory refill or a replica.

L2 Refill Policy

The L2 refill policy looks to replace the following three classes of blocks in descending priority order: (1) an invalid block; (2) a global block with no sharers or an existing replica block; and (3) a global block with active remote sharers.

L1 Eviction Policy

The L1 eviction policy is similar to the L2 refill policy. However, the key observation here is that we *never* want to evict a global block with remote L1 sharers in favor of a local replica, as an actively cached global block is likely to be in the current working set. Therefore, the L1 eviction policy will replace the following two classes of cache blocks in the target set in descending priority order: (1) an invalid block; and (2) a global block with no sharers or an existing replica block. If no blocks belong to any of these two categories, a replica is not made and the victim is evicted from the tile similar to the baseline shared design. Finally, victim replication never creates a victim replica when the home tile happens to be local.

Traditionally in uniprocessors, the replacement policies utilize some form of time-base information, such as LRU. In our simulations, however, we have found that utilizing time-based information did not particularly help with miss rates for the L2 cache. We believe this is because the view of recency from local L2 accesses is not an accurate description of access patterns of the processor. For example, a heavily accessed block in L1 will not generate any local L2 traffic, but it should not be evicted from the L2 cache. Thus, if multiple blocks are available in each category, we simply choose one at random.

4.3.3 Implementation Overhead

Victim replication has a small area overhead over the shared design because the L2 tag must be wide enough to hold physical addresses from any tile. Thus the tag width becomes the same as the private design as shown in Figure 4-4. Global L2 blocks redundantly set these bits to the address index of the home tile. Replicas of remote blocks can be distinguished from regular L2 blocks as their additional tag bits do not match the local tile index.

4.4 Victim Migration

The main limitation of victim replication is that for each shared cache block, a copy must also be present in the L2 cache of its home tile. For multi-threaded applications with a reasonable amount of sharing among threads, this overhead is small. However, for multi-programmed workloads, this data duplication significantly reduces on-chip capacity because the sharing among the different threads is minimal. For these workloads, we expect a pure private design will usually outperform victim replication.

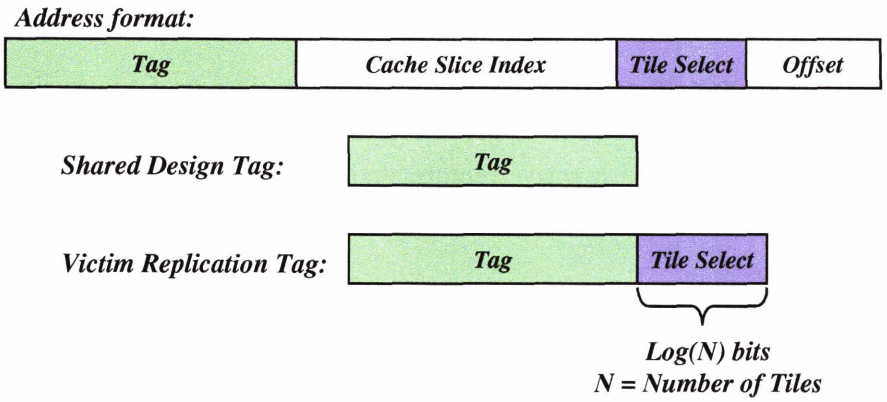


Figure 4-4: The tag width in victim replication is wider than the shared design by $\lg(N)$ bits, where N is the number of tiles in the system. The extra bits are used to distinguish the actual home tile of the address.

Because multi-programmed workloads are expected to be an important component of the workload seen by future systems, we devised victim migration to combat this data duplication problem. Victim migration uses the replication idea, but is more flexible and can dynamically mimic the behavior of a pure private design. Figure 4-5 shows its cache hierarchy arrangement. Each L2 cache consists of a tag array, a data array, and directory bits, similar to the shared design. In addition, each L2 cache also has an extra tag array, which we refer to as the *VM tag array*. To simplify the initial discussion, we assume that the size and associativity of the VM tag array is identical to that of the regular L2 tag array.

4.4.1 Mechanisms

In victim migration, each cache block is held in one of two forms. First, it can be managed exactly like the shared design. Second, if the block is being actively shared by another tile, either as a regular L1 cache block or as an L2 replica, the L2 cache may choose to store only its tag but no data in the VM tag array. By doing so, victim migration removes the unnecessary duplication of data at the home tile, freeing up data array space to hold more replicas or other global blocks. The only added complexity is that both regular and VM tag arrays must be searched during a data fetch. If a hit is found in the VM tag array, the request is satisfied through a three-way cache-to-cache transfer.

4.4.2 Management Policies

We again provide a set of heuristics to efficiently manage the cache on-chip capacity. Specifically, we discuss three policies. The *L2 refill policy* and the *L1 eviction policy* used in victim replication must be retooled to take advantage of the VM tag array. In addition, if the local L2 slice decides not to replicate an L1 victim and sends it back to the home tile, or if a

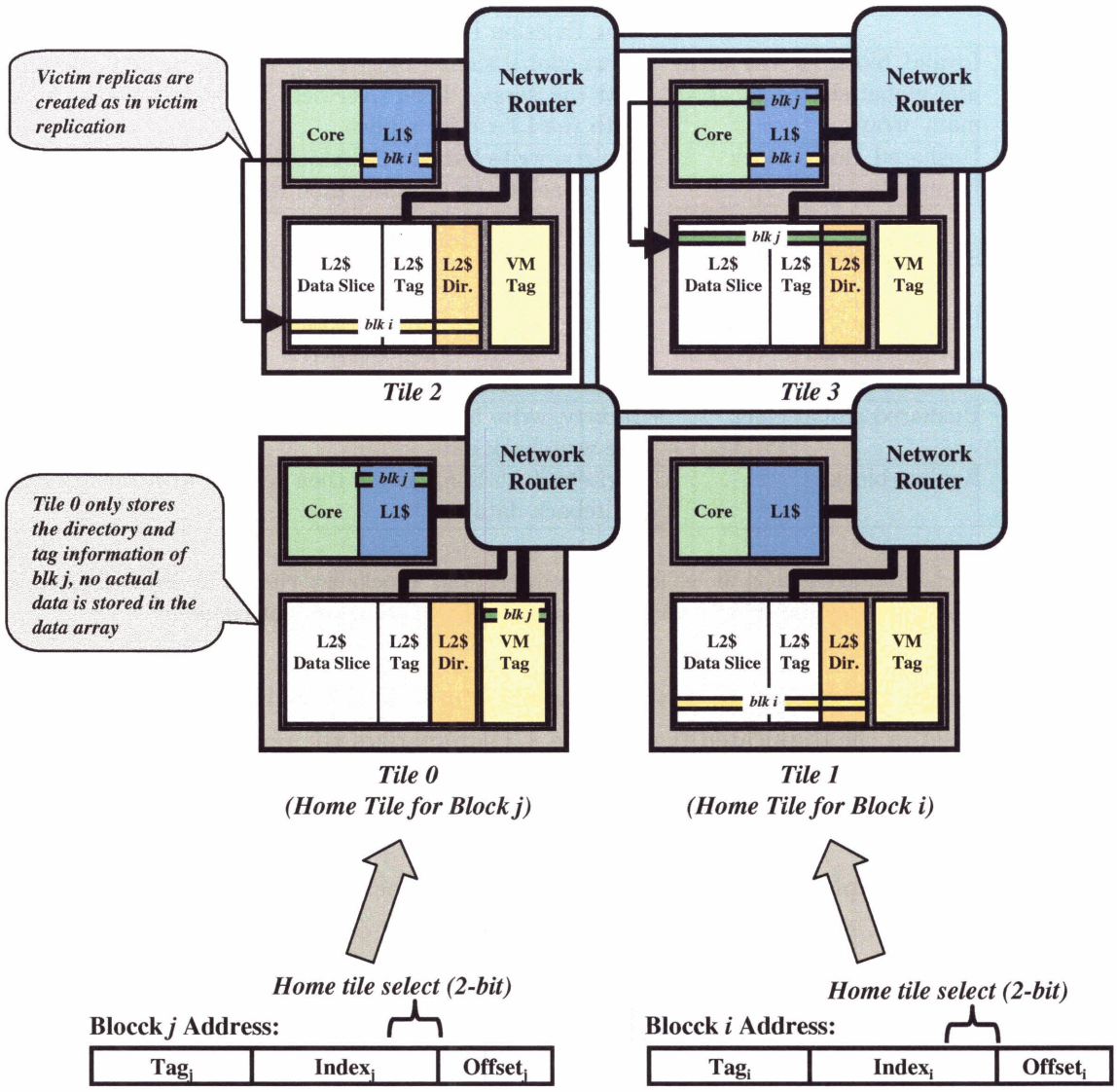


Figure 4-5: *Victim migration* is based on victim replication but more flexible. By using the VM tag array, victim migration removes the unnecessary duplication of data at the home tile, freeing up space to hold more replicas or other global blocks. If a hit is found in the VM tag array, the request is satisfied through three-way cache-to-cache transfers using reply-forwarding.

Priority	Target Block Type(s)	Action
L2 Cache Refill Policy		
1	Invalid block in main array	Refill
	Invalid block in VM array	Refill
2	Unshared global block	If dirty, write back to DRAM, then refill
	Replica block	Writeback to home node, then refill
3	Shared global block	Invalidate all sharers, write back if dirty, then refill
L1 Cache Eviction Policy		
1	Invalid block in VM array *and* global shared block in main array	Move global shared block's tag into the invalid space in VM tag array. Then overwrite the global share block with the L1 victim replica.
2	Unshared global block	If dirty, write back to DRAM, then replace with replica
	Replica block	Writeback to home node, then replace with new replica
Remote Tile Writeback Policy		
1	Shared global block	Swap tags with VM tag entry, overwrite the data with the remote tile writeback data.
2	Invalid block in VM array *and* global shared block in main array	Move global shared block's tag into the invalid space in VM tag array. Then overwrite the global share block with the remote tile writeback data.
3	Unshared global block	If dirty, write back to DRAM, then replace with remote tile writeback data
	Replica block	Writeback to home node, then replace with remote tile writeback data.

Table 4.2: Cache management policies for victim migration. Blocks are chosen in descending order according to their priority and blocks with the same priorities are chosen at random.

replica is evicted, the *remote tile writeback policy* is used to determine where to place the data if it is held in the duplicated tags. Table 4.2 summarizes the policies.

L2 Refill Policy

The L2 refill policy replaces the following three classes of blocks in descending priority order: (1) an invalid block, either in the main tag and data array or in the VM tag array; (2) a global block with no sharers or an existing replica block; and (3) a global block with active remote sharers.

L1 Eviction Policy

The L1 eviction policy determines whether to replicate an L1 victim, and if so, where to hold it in the local L2 slice. We first simultaneously search for an invalid VM tag and an actively shared block in the regular tag array. If both exist, the tag of the actively shared block can be moved to the invalid VM tag entry without losing information. The L1 victim can safely overwrite the shared block's local data. As no data is evicted from the local L2 cache, this operation should not cause performance degradation. The only minor effect may come from the possibly longer hit latency required to perform a three-way cache-to-cache transfer when a remote request hits in the VM tag array and the block was previously stored

in the regular tag array.

If the above scenario is not possible and we must evict a valid block, we look to replace either a global block with no L1 sharers or an existing replica block. If neither of the two exist, we do not replicate the L1 victim. This approach is the same as victim replication.

Remote Tile Writeback Policy

This policy is used whenever a tile has to evict a block back to the home node, either from its primary cache when no replica can be created, or from the victim replicas when they are evicted. At the home tile, if the block is already held in the regular tag and data array, we perform a conventional update. If the tag is held in VM tag array and another tile still has a copy of the data, we simply update the directory information in the VM tag. However, if the last on-chip copy of a cache block is sent home and its tag is kept in the VM tag array, we must decide if and where to keep this unique copy.

We first look for an actively shared global block, which currently does not need the data array space. This global block can be swapped with the remote writeback. If we can find such a swap, no data is evicted from the chip.

If this scenario is not possible, we use the approach outlined in the L1 eviction policy to look for unowned blocks or replica blocks to replace. If a replica is replaced, there can be a ripple effect as the evicted replica is written back to its own home tile.

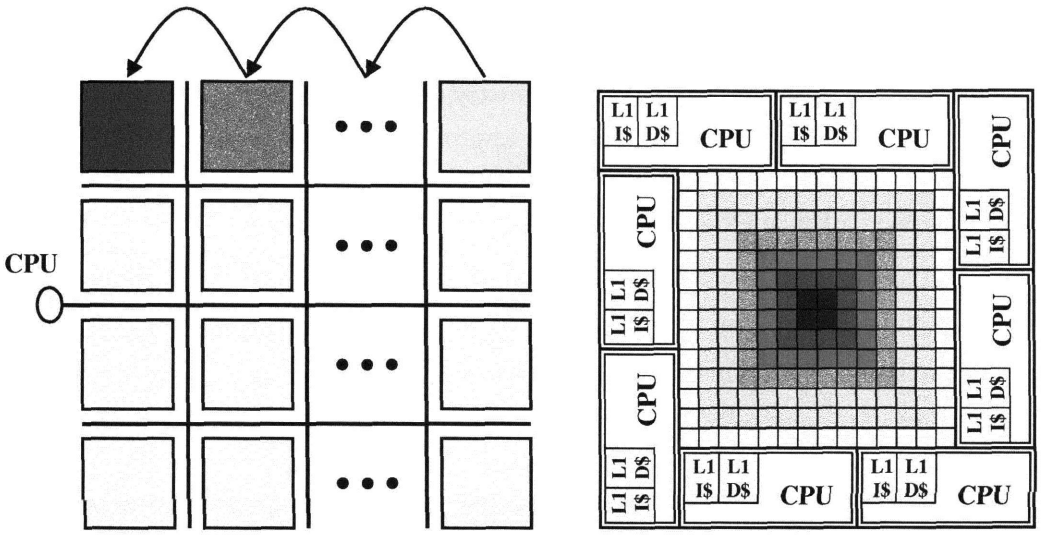
If no unowned blocks or replicas are found, we again choose not to evict actively shared blocks as they are likely to be in the active working set. In this case, the remote tile writeback is evicted from the chip and written back to memory if necessary.

4.4.3 Implementation Overhead

The main drawback of victim migration is its area overhead. First, because victim migration builds upon victim replication, its tag width must be that of the private design. In addition, the VM tag array also keeps the L2 tag and L1 sharing information, which can incur a costly area overhead. In Chapter 6, we will show that the size of the VM tag array can be reduced to to one fourth of the regular L2 tag array size and still achieve reasonable latency reduction. The overall area used by such a design is smaller than the private design. This is because the private design must also use duplicated set of L2 tag arrays to implement the on-chip directory, incurring a significant area overhead.

4.5 Related Work

A number of proposals seek to reduce the effective access latency of a large shared cache by adopting a non-uniform cache access (NUCA) architecture. NUCA [KBK02] designs allow access latency to vary depending on the relative placement of the processor and L2 slice containing the data. Dynamic NUCA designs have been proposed for uniprocessors [KBK02,



(a) Data migration in uniprocessor: D- NUCA. ***Source: Kim et. al. ASPLOS-X, 2002.***

(b) Data migration in chip multiprocessor. ***Source: Beckmann et. al. MICRO-37, 2004.***

Figure 4-6: Examples of data migration. Each rectangle represents a cache slice, with the darker squares representing rectangles slices that are accessed more frequently. Figure(a) shows D-NUCA [KBK02], a scheme that dynamically moves the more frequently used data to the closer slices to the processor core. Figure(b) shows a data migration study conducted in [BW04] on a CMP. The study shows that data migration might not work well as shared data tend to migrate to locations equidistant to all sharers. In the configuration shown here, all shared data moves to the center of the chip.

CPV03], where frequently-accessed cache blocks gradually migrate closer to the processor. Figure 4-6(a) shows this approach taken by [KBK02]. These schemes are considerably more complicated when applied to CMPs with the "dance-hall" configurations [BW04, CPV05, HKS+05] discussed in Chapter 3. They require some form of duplicated L2 tag array kept local to each processor to reduce the number of slices that must be searched to locate an on-chip block. Further, all such local tag arrays must be kept consistent with any block migration triggered by a remote processor, imposing additional serialization constraints on otherwise independent cache accesses [BW04, CPV05, HKS+05].

Data migration techniques [KBK02, CPV03] discussed in the introduction could have poor performance when applied to tiled CMPs because a given L2 block may be repeatedly accessed by processor cores at opposite corners of the die. A recent study [BW04] investigates the behavior of block migration in CMPs using a variant of D-NUCA, but the proposed protocol is complex and relies on a "smart search" algorithm for which no practical implementation is given. The benefits are also limited by the tendency for shared data to migrate to the center of the die. This phenomenon is shown in Figure 4-6(b).

Several proposals advocate data replication [CPV05, SSZR05], which allow sharers to replicate local copies of shared data for fast access. CMP-NuRAPID [CPV05] extends NuRAPID to support data replication for CMPs based on a snooping coherence protocol.

The actual implementation, however, is complex and incurs a large area overhead. In the baseline IBM Power4 scheme [TDJ+02], each node has a non-inclusive L3 cache that stores the local L2 victims. However, while L3s can be snooped by other nodes, the local L2 victim always overwrites the local L3, causing considerable pressure on the L3 and reducing the effective L3 capacity. In [SSZR05], this baseline design is improved by using a small history table to selectively remove some clean writebacks of data already present in the L3 cache.

Data replication also bears resemblance to earlier work on remote data caching in conventional CC-NUMA and COMA architectures [OR99, DT99, ZT97], which also try to retain local copies of data that would otherwise require a remote access. There are two major differences in the CMP structure, however, that limit the applicability of prior remote caching work. First, in CC-NUMAs, all of the local cache capacity on a node is private so the allocation between local and remote data only affects the local node. In a CMP, on-chip L2 capacity is shared by all nodes, and so a local node's replacement policy affects cache performance of all nodes. Second, in both CC-NUMA and COMA systems, remote data is further away than local DRAM, thus it is beneficial to use a large remote cache held in local DRAM. In addition, the cost of adding a remote cache is low and does not diminish the performance of existing L2 caches. In the CMP structure, the remote caches are closer to the local node than any DRAM, and any replication reduces the effective cache capacity for blocks that will have to be fetched from slow off-chip memory.

Chapter 5

Experimental Methodology

To evaluate the various cache management policies, we implemented a flexible and detailed cache-coherent distributed shared memory system model that includes L1 caches, L2 caches, main memory, and an interconnection network. In this chapter, we describe our simulation infrastructure and techniques, and provide discussions on the choice of the system parameters and the workload suite.

5.1 Simulation Infrastructure

To present a clearer picture of the memory system's behavior, we use a simple in-order processor model and focus on the average raw memory latency seen by each memory request. Clearly, overall system performance can only be determined by co-simulation with a detailed processor model, though we expect the performance trend to closely follow average data access latency. Prefetching, decoupling, non-blocking caches, multi-threading, and out-of-order execution are well-known microarchitectural techniques which overlap memory latencies to reduce their impact on performance. However, machines using these techniques complete instructions faster, and are therefore relatively more sensitive to any latencies that cannot be hidden. Also, these techniques are complementary to our replication techniques, and cannot provide the same benefit of reducing cross-chip traffic.

5.1.1 Simulator Setup

We have implemented a full-system execution-driven simulator based on the Bochs [Law] system emulator. We added a cycle-accurate cache and memory simulator with detailed models of the primary caches, the L2 caches, the 2D mesh network, and the DRAM. Although the combined limitations of Bochs and our Linux port restrict our simulations to eight processor cores, it's full-system nature allows us to run realistic workloads that require operating system support. Furthermore, the execution-driven nature of the simulator allows our detailed memory system to affect the interleaving of threads, which is difficult to

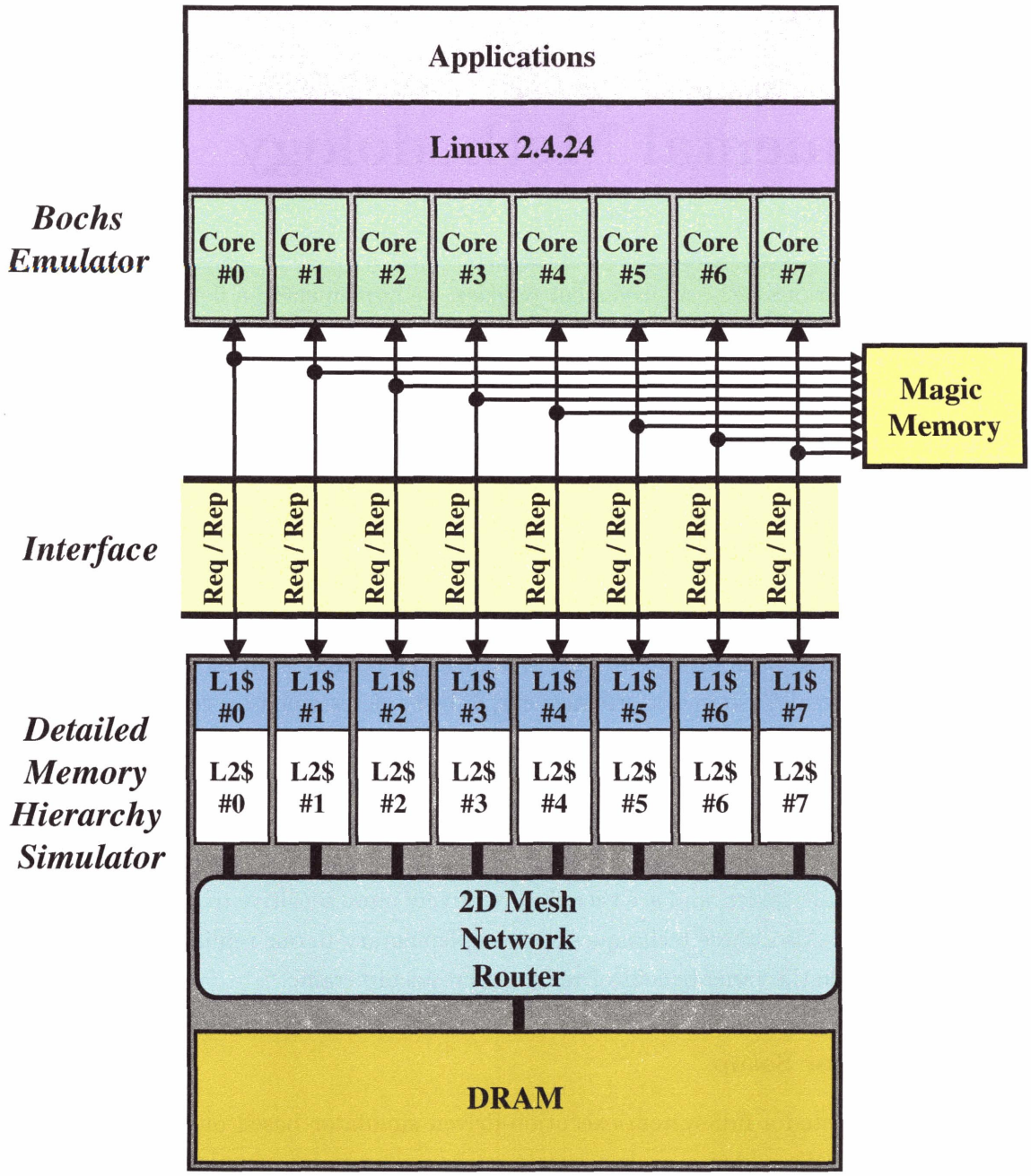


Figure 5-1: The overall simulation infrastructure. A detailed cache and memory simulator is developed to experiment with the cache designs. The Bochs full-system emulator is used as the processor model and drives the detailed cache and memory simulator to form an execution-driven system simulator.

achieve in trace-driven simulation. The workloads are compiled under Linux version 2.4.24. This version of Linux is compiled for an x86 processor on an eight-way SMP. The overall simulator infrastructure is shown in Figure 5-1. The detailed memory model consists of three parts: the L1 and L2 caches for each tile, the DRAM module, and the interconnection network as described in Chapter 3. The memory references from the code sequence are extracted and fed into the detailed memory model, which provides a request-response interface to the Bochs emulator. The execution rate of each processor in the Bochs emulator is controlled by the feedback from the memory system. The magic memory is used to provide values for processor data accesses during the fastforwarding phase, which we will describe in Section 5.3.

5.1.2 Interfacing Bochs to Detailed Cache Simulator

Figure 5-2 details the interface between the Bochs emulator and our detailed simulator. The main loop of Bochs moves round-robin between the processor cores, I/O devices, and disk, incrementing the cycle count at the end of each loop. Devices that need attention assert an interrupt line and are handled by the operating system of the simulated machine. During each "Bochs cycle", one x86 instruction is executed. All instructions that do not contain memory accesses are executed normally. For instructions that invoke memory accesses, we extract these accesses and feed them into detailed memory simulator.

Because Bochs is an x86 architecture emulator, a single instruction could touch memory multiple times, such as the `lea` example in processor core *1* or the **pusha** example in processor core *2*. To simulate such an instruction accurately, we ought to suspend the processor execution appropriately for each memory access that does not hit the cache, until it is resolved by the memory simulator, then continue onto the next memory access. However, this approach requires significant modifications to the Bochs emulator to implement a mechanism that checkpoints the state of the simulator in the middle of an instruction. The frequent checkpointing could also significantly impact running time.

To avoid this cumbersome overhaul to Bochs, we took a simpler approach to handle instructions with multiple memory accesses. Such an instruction is executed to completion, performing all of the memory accesses necessary using data from the magic memory. The actual loads and stores are buffered up in a *memory access buffer* and forwarded to the detailed memory system. Execution for the requesting processor is suspended until all of the memory accesses are resolved. Checkpointing here can only happen in between instructions.

Our approach can create complications for a subset of the instructions with multiple memory accesses when the address of a later fetch depends on the result of an earlier fetch. Because of this dependence, coupled with the memory simulator timing and the specific thread interleaving, the values fetched from the memory simulator may deviate from the ones provided by magic memory. In actual simulation, we have found such instances to be

rare (under 1%). When it does happen, however, we use a fixup mechanism to force the data in the memory simulator to match the magic memory, so that future memory accesses to the memory simulator can produce the same values as the magic memory. This approach guarantees that we are executing the workload with a legal thread interleaving.

5.1.3 Simulation Parameters

In this thesis, we chose to simulate four cache configurations. The parameters of each configuration are summarized in Table 5.1. To simplify result reporting, we scaled all system latencies to the access time of the L1 cache, which we assume can be reached within a single clock cycle.

We picked the 70 nm technology parameters based on the Berkeley Predictive Technology Model (BPTM) [UC 01]. We use a 16 FO4 clock cycle [Hor83] time for configuration 1 because it has a smaller 16KB L1 cache. We assume a 24 FO4 clock cycle time for Configurations 2 through 4 because they have a larger 32KB L1 cache. Both 16 FO4 and 24 FO4 cycle times represent modern power-performance balanced pipeline designs [HP03, SBG+02]. High-frequency designs may target cycle times of 8 FO4 to 12 FO4 delays [HBJ+02, SC02], in which case our cycle latencies can be scaled appropriately. A five-cycle access latency is used for a 256KB L2 cache with a six-cycle latency for 512KB and 1MB caches. We also scale all other latencies appropriately for the smaller Configuration 1. Specifically, assuming the same absolute off-chip fetch latency, the relative latency of the DRAM in this configuration is significantly longer than the other three, at 192 cycles.

We model each hop in the network as taking 3 cycles, including the router latency and an optimally-buffered inter-tile copper wire on a high metal layer. Note that the worst case contention-free L2 hit latency is between 29 to 32 cycles for these configurations, hinting that even a small reduction in cross-chip accesses could lead to a significant performance gain. The 16-way L2 set-associativity was chosen to be larger than the number of tiles, thus avoiding most of the cache thrashing caused by different threads. In our simulations, we have found that for L2 associativities of eight or less, several workloads had severe inter-thread conflicts, reflected by high off-chip miss rates.

5.2 Workloads

This section summarizes the collection of workloads used to evaluate the cache management policies. To minimize system variability, all workloads were invoked in a runlevel without superfluous processes/daemons to prevent non-essential processes from interfering with the workload execution. Each simulation run begins with the Linux boot sequence, but results are only gathered after the workload begins execution until its completion.

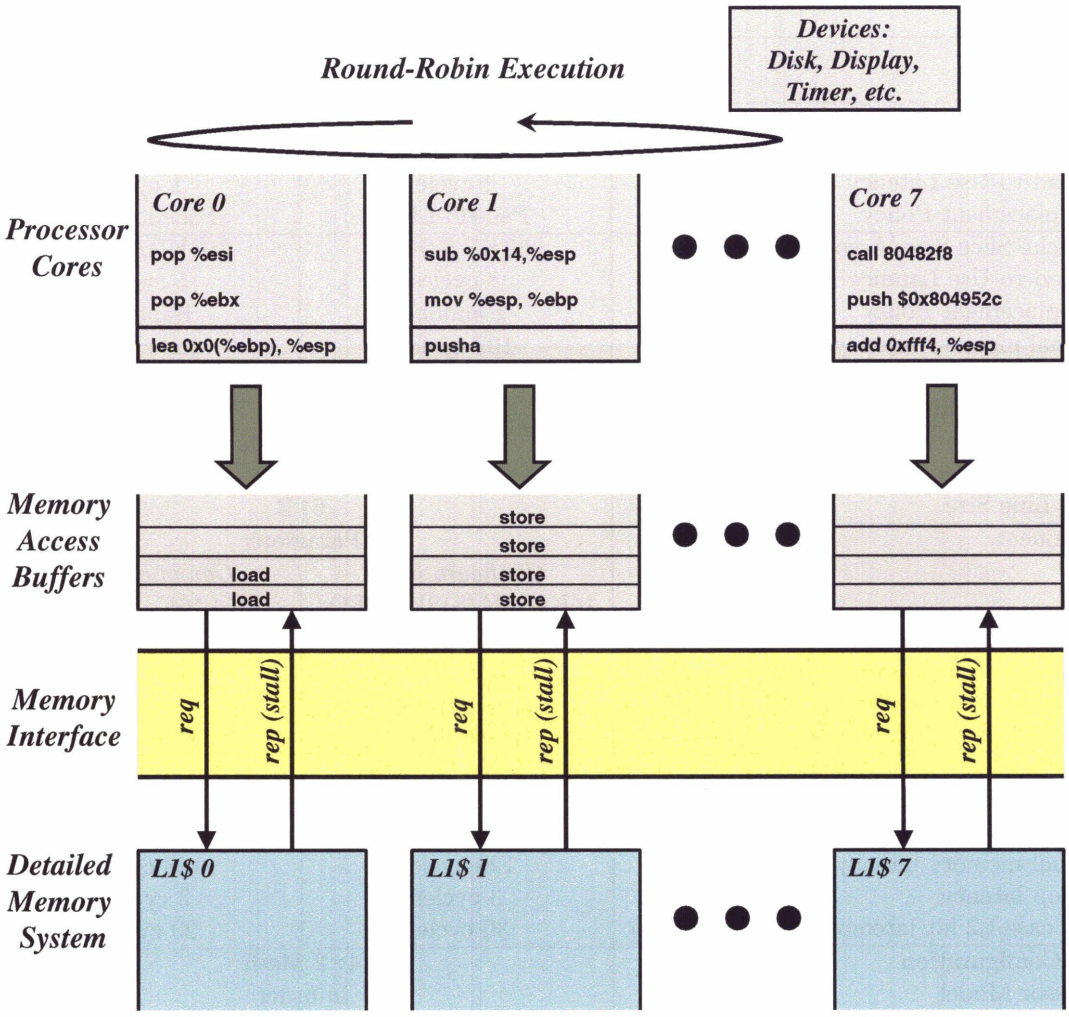


Figure 5-2: Illustration of the execution-driven model combining the Bochs emulator with the detailed memory system. The data and instruction access streams in each instruction are buffered in a data access buffer and fed to the memory simulator. The access results are fed back to the simulator to control the progress of execution.

Component	Parameter	
	Configuration 1 8K+8K/256K/16FO4	*Configuration 2* 16K+16K/256K/24FO4
L1 I-Cache Size/Associativity	8 KB/16-way	16 KB/16-way
L1 D-Cache Size/Associativity	8 KB/16-way	16 KB/16-way
L1 Load-to-Use Latency	1 cycle	1 cycle
L1 Replacement Policy	Psuedo-LRU	Psuedo-LRU
L2 Cache Slice Size/Associativity	256 KB/16-way	256 KB/16-way
L2 Load-to-Use Latency (per slice)	8 cycles	5 cycles
L2 Replacement Policy	Random	Random
External memory latency	192 cycles	128 cycles
One-hop latency	3 cycles	3 cycles
Worst case L2 hit latency (contention-free)	32 cycles	29 cycles
CMP Configuration	4×2 Mesh	
Processor Model	in-order	
Cache Line Size	64 B	
Component	Parameter	
	Configuration 3 16K+16K/512K/24FO4	*Configuration 4* 16K+16K/1M/24FO4
L1 I-Cache Size/Associativity	16 KB/16-way	16 KB/16-way
L1 D-Cache Size/Associativity	16 KB/16-way	16 KB/16-way
L1 Load-to-Use Latency	1 cycle	1 cycle
L1 Replacement Policy	Psuedo-LRU	Psuedo-LRU
L2 Cache Slice Size/Associativity	512 KB/16-way	1 MB/16-way
L2 Load-to-Use Latency (per slice)	6 cycles	6 cycles
L2 Replacement Policy	Random	Random
External memory latency	128 cycles	128 cycles
One-hop latency	3 cycles	3 cycles
Worst case L2 hit latency (contention-free)	30 cycles	30 cycles
CMP Configuration	4×2 Mesh	
Processor Model	in-order	
Cache Line Size	64 B	

Table 5.1: Simulation parameters. The numbers for each configuration represent the cache sizes and cycle times. For example, 8K+8K/256K/16F04 indicates 8KB L1 instruction cache, 8KB L1 data cache, 256KB L2 cache, with a 16 FO4-delay cycle time.

Workload Name	Instruction (Billions)	Workload Description
bzip2	3.8	Based on the popular `bzip2` compression algorithm version 0.1
crafty	1.2	A high-performance chess program designed around a 64-bit word.
eon	2.9	A probabilistic ray tracer based on Kajiya's 1986 SIGGRAPH paper.
gap	1.1	Gap implements a language and library designed mostly for computing in groups (GAP is an acronym for Groups, Algorithms and Programming).
gcc	6.4	gcc is based on `gcc` version 2.7.2.2. It generates code for a Motorola 88100 processor. The benchmark runs as a compiler with many of its optimization flags enabled.
gzip	1.0	gzip (GNU zip) is a popular data compression program written by Jean-Loup Gailly for the GNU project. `gzip` uses Lempel-Ziv coding (LZ77) as its compression algorithm.
mcf	1.7	A benchmark derived from a program used for single-depot vehicle scheduling in public mass transportation. The program is written in C, the benchmark version uses almost exclusively integer arithmetic.
parser	5.6	The Link Grammer Parser is a syntactic parser of English, based on link grammer, an original theory of English syntax. Given a sentence, the system assigns it a syntactic structure, which consists of set of labeled links connecting pairs of words.
perlbmk	1.8	perlbmk is a cut-down version of Perl v5.005_03, the popular scripting language.
twolf	1.5	The TimberWolfSC placement and routing CAD tool package.
vortex	1.5	VORTEx is a single-user object-oriented database transaction benchmark which which exercises a system kernel coded in integer C. The benchmark vortex is a subset of a full object oriented database program called VORTEx (Virtual Object Runtime EXpository).
vpr	5.3	VPR is a placement and routing program; it automatically implements a technology-mapped circuit (i.e. a netlist, or hypergraph, composed of FPGA logic blocks and I/O pads and their required connections) in a Field-Programmable Gate Array (FPGA) chip.

Table 5.2: Single-threaded workloads in this thesis are taken from the SpecINT2000 benchmark suite [Cor00].

Workload Name	Instruction (Billions)	Workload Description
NAS Scientific Applications		
BT	1.7	A simulated CFD application that uses an alternating direction implicit (ADI) approximate factorization to solve 3D compressible Navier-Strokes equations. Class S.
CG	5.0	Computation of an approximation to the smallest eigenvalue of a large, sparse, unstructured matrix using the conjugate gradient method. Class W.
EP	6.8	An embarrassingly parallel benchmark. It generates pairs of Gaussian random deviates. Class W.
FT	6.6	The computational kernel of a 3D Fast Fourier transform (FFT)-based spectral method. FT performs three one-dimensional FFTs, one per dimesion. Class S, -O0.
IS	5.5	Integer sort. Class W, compiled using icc-v8.
LU	6.2	LU decomposition that uses symmetric successive over-relocatoin (SSOR) method to solve a seven-block-diagonal system. Class R.
MG	5.1	MG uses a V-cycle multigrid method to compute an approximation to the smallest eigenvalues of a large, sparce, unstructured matrix. Class W.
SP	6.7	A simulated CFD application that uses a Beam-Warming implicit (ADI) approximate factorization to solve 3D compressible Navier-Strokes equations. Class R.
System Applications		
apache	3.3	Apache benchmark's 'ab' worker threading model, 2000 requests, 3 at a time. Compiled with gcc 2.96.
dbench	3.3	Executes Samba-like syscalls, 3 clients, 10000 requests. Compiled with gcc 2.96.
AI Application		
checkers	2.9	Cilk checkers (parallel $\alpha - \beta$ search), Black plies 6, White plies 5. Compiled using Cilk 5.3.2 and gcc 2.96.

Table 5.3: Multi-threaded workloads include the NAS parallel scientific benchmark suite, two system workloads, and one AI application [Gro01, BBB+94].

Workload Name	Instruction (Billions)	Workload Description
mix0	23.9	bzip, crafty, eon, gap, gcc, gzip, mcf, and parser
mix1	24.8	gcc, gzip, mcf, parser, perlbmk, twolf, vortex, and vpr
mix2	19.1	bzip, crafty, eon, gap, perlbmk, twolf, vortex, and vpr
mix3	22.8	bzip, gap, mcf, twolf, crafty, gcc, parser, and vortex
mix4	19.1	bzip, gap, mcf, twolf, eon, gzip, perlbmk, and vpr
mix5	25.7	crafty, gcc, parser, vortex, eon, gzip, perlbmk, and vpr
mix6	12.7	crafty, eon, gap, gzip, mcf, perlbmk, twolf, and vortex
mix7	21.5	bzip2, gap, gzip, mcf, parser, twolf, vortex, and vpr
mix8	28.0	bzip2, crafty, eon, gap, gcc, mcf, parser, and vpr

Table 5.4: Multi-programmed workloads are created by mixing single-threaded benchmarks. Eight benchmarks are randomly chosen for each multi-programmed workload.

5.2.1 Single-Threaded Workloads

For single-threaded workloads, we used all twelve benchmarks in the SpecINT2000 benchmark suite, summarized in Table 5.2. They are compiled with the Intel C compiler (version 8.0.055) using `-O3 -static -ipo -mp1 +FDO` and use the MinneSPEC large-reduced dataset as input. The size of the workloads ranges from one billion cycles to over six billion cycles.

5.2.2 Multi-Threaded Workloads

The multi-threaded workloads include all eight of the OpenMP NAS Parallel Benchmarks (NPB) (mostly written in FORTRAN), two server workloads (written in C), and one AI workload (written in Cilk [Gro01]). Table 5.3 summarizes the workloads. For the NAS Parallel Benchmarks, classes S and W are standard input sizes, and class R is custom-sized to give the workload a manageable simulation time that falls between the S and W classes. The two server benchmarks, `apache` and `dbench`, spend significant execution time in the operating system. Additionally, one AI benchmark, `checkers`, uses a dynamic work-stealing thread scheduler. All of the multi-threaded benchmarks are compiled with `ifort-v8 -g -O2 -openmp` unless otherwise noted. The size of the workloads range from 1.7 billion to 6.8 billion instructions.

5.2.3 Multi-Programmed Workloads

The multi-programmed workloads are created by mixing a set of randomly selected single-threaded SpecINT2000 workloads, each consisting of eight different programs. Therefore, the size of the workloads are much larger than that of the single-threaded and multi-threaded workloads, generally at around twenty billion instructions.

5.3 Fastforwarding Multiprocessor Simulation

Due to the long running nature of our workloads, we used a sampling technique to reduce workload simulation time. Figure 5-3 illustrates several traditional approaches in speeding up simulation.

Many architecture studies have obliviously chosen a single sample, either taken from the beginning or after some fixed number of instructions into the run, as shown in Figure 5-3(a) and Figure 5-3(b). A detailed warm-up phase preceding the actual data gathering phase can warm-up large data structures such as the branch predictor and caches, thus give more accurate results (Figure 5-3(c)). A better approach is to search for an execution phase that is representative of the workload's overall characteristics through profiling, and only gather data in this representative phase.

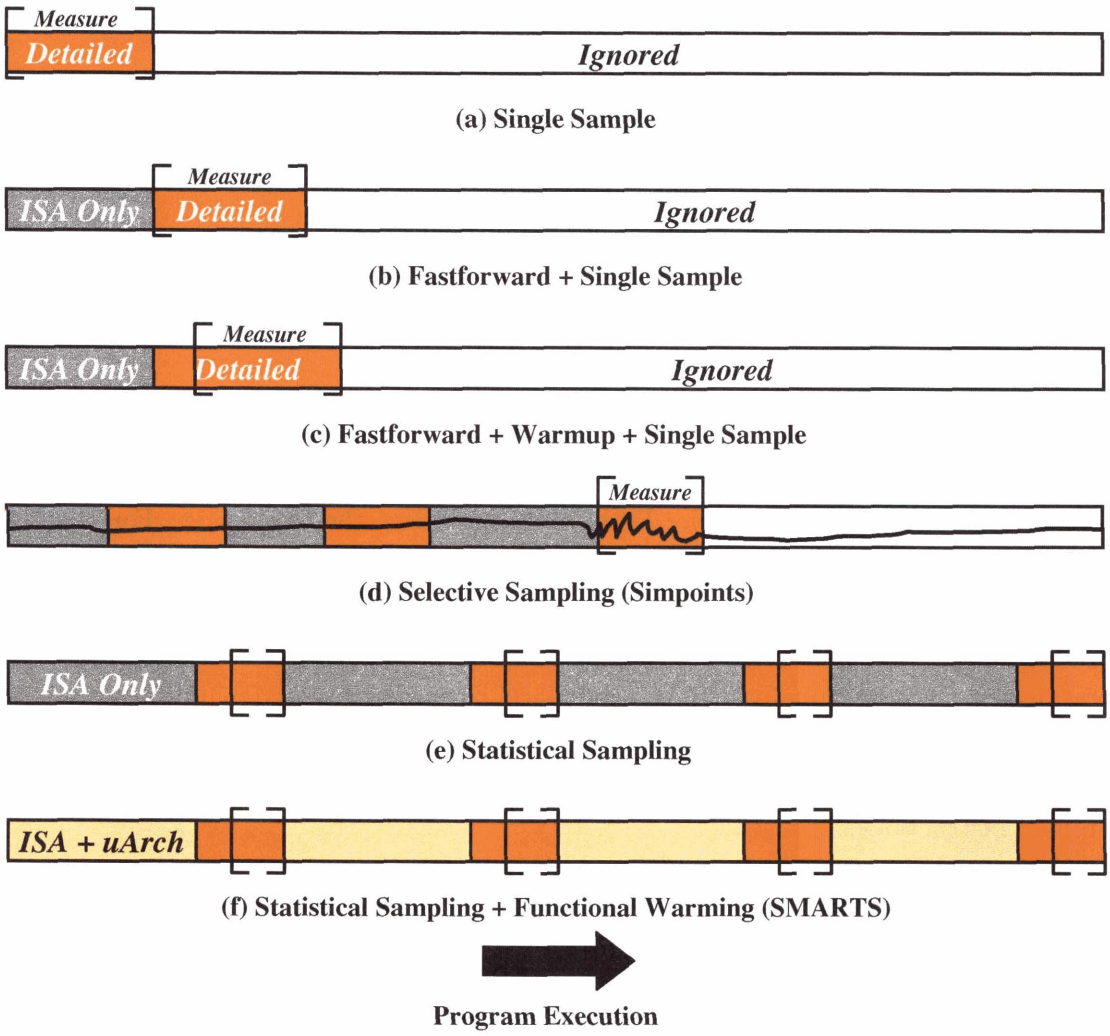


Figure 5-3: (a) Statistics gathering in a single sample at the beginning of the execution. (b) Statistics gathering in a single sample in the middle of the execution after initial fastforwarding. (c) Statistics gathering is preceded by fastforwarding and detailed warming. (d) A representative sample determined by profiling is used over a random sample. (e) Repetitive statistical sampling with multiple sample points. (f) Functional warming is used to minimize the detailed warming phase.

However, applications generally contain multiple phases of execution with varying properties and much better characterization is possible by using multiple sample points spread throughout a run. Statistical sampling [SPHC02, CHM96, LPI88] uses an ISA simulator during the fastforwarding phase, then constructs the architecture state through detailed warming before actually gathering data. To minimize the warming phase for architecture with large amounts of state, SMARTS [WWFH03] uses a functional simulator during the fastforwarding phase to update the architectural state, such as registers and memory, then switches to a slower detailed simulator to accurately model the microarchitecture during the measurement samples.

In this thesis, we extend the functional warming method for superscalars proposed in SMARTS [WWFH03] to an SMP system, and fastforward through periods of execution while maintaining cache and directory state [BPZA05]. In fastforwarding mode, we do not forward the load and store requests to the detailed memory timing simulator, but only update cache and directory state fields. At the start of each measurement sample, we run the detailed timing model to warm up the pipelines of the cache (the detailed warming phase), memory and network. After this detailed warming phase, we gather detailed statistics for one million instructions, before re-entering fastforward mode. Detailed samples are taken at random intervals during execution and include 20% of all instructions executed, i.e., fastforward intervals average around five million instructions. The number of samples taken for each workload ranges from around 150 to 1,000. Simulations show that the fastforwarding results match up with detailed runs to within 3% of error.

5.3.1 System Variability

Because we are running multi-threaded application with the operating system, our simulation results are more vulnerable to system variability than uniprocessor simulations. As Alameldeen and Wood point out, even with small variation in DRAM latencies, the overall system result can be noticeably affected [AW03]. To minimize the bias in the results created by this variation, we chose to execute multiple runs of each workload with varying sample length and frequency. We found that the variability has an insignificant effect on our results for these workloads.

Chapter 6

Experimental Results

This chapter presents the results of evaluating the four cache designs in this thesis on our workload suite. We show that victim replication and victim migration provide better and significantly more robust performance than the standard techniques. We first present the results for the multi-threaded workloads because they best demonstrate the major trade-offs in our management policies, then we move on to the results for single-threaded and multi-programmed workloads.

For each class of applications, we show several results. First, we show the average memory access latency seen by a processor. This the the key metric that we aim to minimize. Second, to better understand the trade-offs outlined in Chapter 4, we also show a breakdown of the accesses by category. Third, we show the percentage of replicas held in the L2 caches, demonstrating that our techniques dynamically exploit the different characteristics of individual benchmarks and their execution phases. Finally, we show that victim replication and victim migration also significantly reduce the on-chip traffic, which is an important factor in reducing system power consumption.

6.1 Multi-Threaded Workloads

Figures 6-1 to 6-4 show the key result, the average memory access latency seen by a processor. The minimum latency is one cycle, when all accesses hit in the L1 cache. In the following, we take Configuration 1 (8KB L1 I-cache, 8KB L1 D-cache, 256KB unified L2-cache, 16FO4 cycle time), and give a detailed analysis of how each of the four designs works.

Figure 6-5 shows the breakdown of memory accesses for victim replication (the breakdown for victim migration is similar). An access in this figure belongs to one of six categories:

1. *L1 hits*: Access results in an L1 cache hit.

2. *Local L2 hits*: Access results in an L2 cache hit in the local slice of the L2 cache. For the private design, all hits are local L2 hits. For the other three designs, local L2 hits

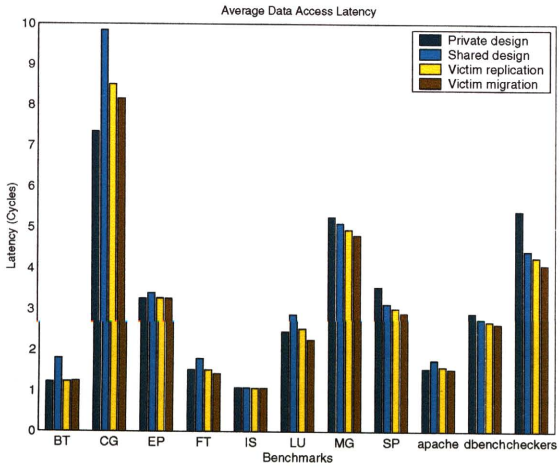


Figure 6-1: Configuration 1: 8KB+8KB/256KB/16FO4. Average access latencies of multi-threaded workloads.

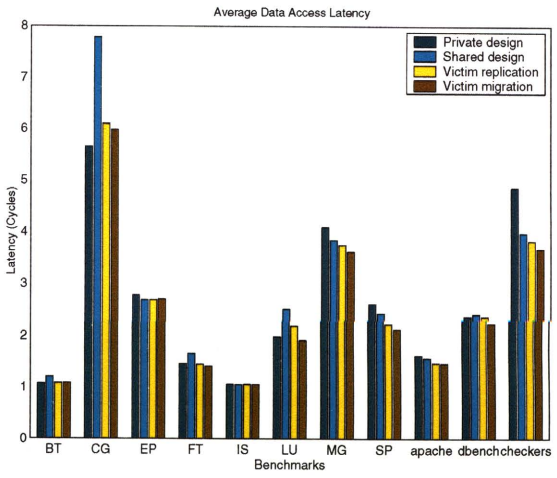


Figure 6-2: Configuration 2: 16KB+16KB/256KB/24FO4. Average access latencies of multi-threaded workloads.

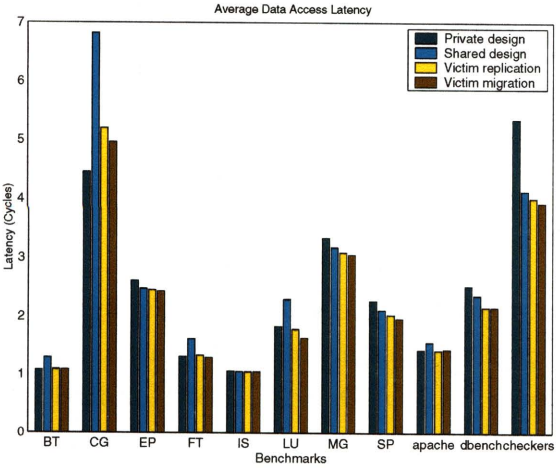


Figure 6-3: Configuration 3: 16KB+16KB/512KB/24FO4. Average access latencies of multi-threaded workloads.

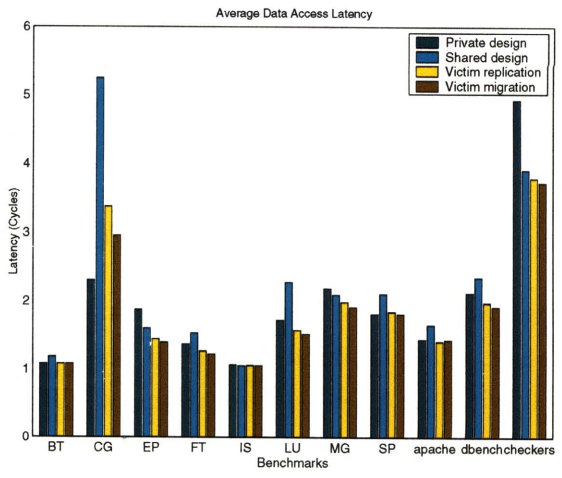


Figure 6-4: Configuration 4: 16KB+16KB/1MB/24FO4. Average access latencies of multi-threaded workloads.

Average Access Latency Reduction of Multi-Threaded Applications

	Configuration 1 8K+8K/256K/16FO4					Configuration 2 16K+16K/256K/24FO4				
	Reduction (%)					Reduction (%)				
Workload	$\frac{VR}{S}$	$\frac{VR}{P}$	$\frac{VM}{S}$	$\frac{VM}{P}$	$\frac{VM}{VR}$	$\frac{VR}{S}$	$\frac{VR}{P}$	$\frac{VM}{S}$	$\frac{VM}{P}$	$\frac{VM}{VR}$
BT	46.1	-1.3	45.3	-1.9	-0.6	11.2	-0.7	11.5	-0.6	0.2
CG	27.7	-4.6	30.3	-2.7	2.0	27.5	-7.3	30.0	-5.6	1.9
EP	3.7	0.0	4.1	0.3	0.3	0.1	3.2	-0.4	2.7	-0.5
FT	18.0	-1.4	26.4	5.6	7.1	14.6	0.9	17.5	3.5	2.5
IS	0.6	0.4	0.8	0.6	0.1	0.0	0.1	-0.0	0.1	0.0
LU	13.3	-2.9	27.3	9.0	12.3	15.0	-9.5	31.6	3.5	14.4
MG	3.2	6.3	6.1	9.3	2.8	2.5	9.1	6.0	12.9	3.4
SP	3.5	17.4	7.7	22.1	4.0	9.1	17.2	14.6	23.1	5.0
apache	10.0	-3.6	14.0	-0.2	3.6	6.3	9.9	7.3	10.9	0.9
dbench	2.2	7.5	4.5	9.9	2.2	2.0	0.37	8.2	6.4	6.0
checkers	3.8	26.5	8.7	32.5	4.7	3.9	26.8	8.1	31.9	4.0
Avg	12.0	4.0	15.9	7.6	3.5	8.4	4.5	12.2	8.1	3.4
	Configuration 3 16K+16K/512K/24FO4					Configuration 4 16K+16K/1M/24FO4				
	Reduction (%)					Reduction (%)				
Workload	$\frac{VR}{S}$	$\frac{VR}{P}$	$\frac{VM}{S}$	$\frac{VM}{P}$	$\frac{VM}{VR}$	$\frac{VR}{S}$	$\frac{VR}{P}$	$\frac{VM}{S}$	$\frac{VM}{P}$	$\frac{VM}{VR}$
BT	18.4	-0.9	18.9	-0.6	0.4	9.4	0.0	9.4	0.0	0.0
CG	31.1	-14.4	37.4	-10.3	4.8	55.2	-31.7	77.3	-22.1	14.2
EP	0.6	6.3	1.8	7.5	1.1	10.4	29.4	14.8	34.5	3.9
FT	20.8	-1.9	25.2	1.6	3.6	21.1	8.1	25.8	12.3	3.8
IS	0.1	1.1	0.3	1.3	0.1	-0.2	0.3	0.2	0.9	0.5
LU	28.4	2.5	40.5	12.2	9.4	44.7	9.6	50.1	13.7	3.7
MG	2.8	7.9	4.1	9.2	1.2	5.0	9.7	9.6	14.5	4.3
SP	4.2	11.8	7.7	15.5	3.3	14.2	-1.7	16.5	0.2	2.0
apache	10.0	0.7	8.8	-0.4	-1.1	17.2	2.3	15.4	0.7	-1.6
dbench	9.2	16.6	9.6	17.0	0.3	18.6	7.3	22.3	10.7	3.1
checkers	3.1	33.3	5.3	36.2	2.1	3.1	29.7	4.9	32.0	1.7
Avg	11.7	5.7	14.5	8.1	2.3	18.1	5.8	22.4	8.9	3.2

Table 6.1: Average access latency reduction of multi-threaded workloads achieved by victim replication and victim migration over the shared and private baseline designs. The five numbers for each workload indicate the percentage reduction of VR to shared, VR to private, VM to shared, VM to private, and VM to VR.

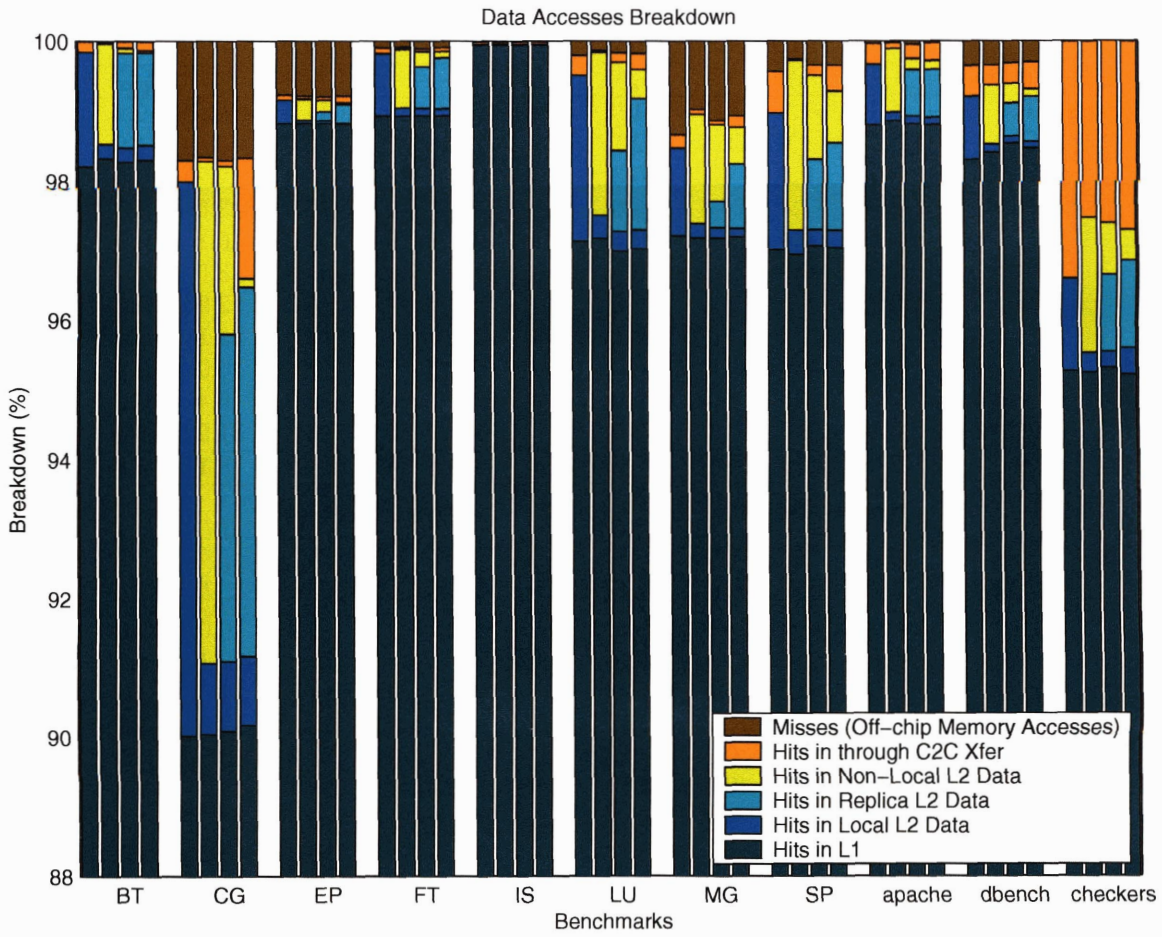


Figure 6-5: Memory access breakdown of multi-threaded workloads. Moving from left to right, the four bars for each workload are for the private design, the shared design, victim replication, and victim migration, respectively. Hits are categorized into (from bottom to top): (1) L1 hits; (2) L2 local hits; (3) replica hits; (4) L2 remote hits; (5) cache-to-cache hits; (6) off-chip accesses.

include all the hits whose home tile happens to be local.

3. *Replica hits*: Access results in a replica hit in the local slice of the L2 cache.

4. *Global L2 hits*: Access results in an L2 cache hit in a remote slice of the L2 cache. The private design cannot have global L2 hits. For the shared design, victim replication, and victim migration, this includes all L2 hits whose home tile is not local.

5. *Cache-to-cache hits*: Access results in an L2 coherence miss, where the requested data is stored in another tile. A cache-to-cache transfer between the owner/sharer of the requested data and the requestor is used to satisfy the request. The transfer happens between L2 caches in the private design and between L1 caches in the other three designs.

6. *Off-chip miss*: Data is not on-chip and the request is forwarded to the off-chip memory.

6.1.1 Performance Analysis

From Figure 6-1, we observe that for the multi-threaded workloads, the performance difference between private and shared designs is significant. We divide their behavior into three scenarios and discuss each separately: (1) equal performance for private and shared designs; (2) private design better than shared design; (3) shared design better than private design.

Equal Performance for Private and Shared Designs

One workload, `IS`, has a working set small enough that it fits in the L1 cache, thus L2 policies do not matter because most data accesses hits in the L1. Average access latency in this case is roughly one cycle for all four policies, equaling the access latency of the L1 cache.

Private Design Better Than Shared Design

Compared to the shared design (second bar in Figure 6-5), the private design (first bar in Figure 6-5) has higher off-chip miss rates, but also many more local hits across the workloads. We expect the private design to win if the difference in the off-chip miss rates is small compared to the extra number of local hits it has over the shared design. This is the case for `BT`, `CG`, `EP`, `FT`, `LU`, and `apache`.

For these workloads, the private design does better than the shared design for two reasons. First, if the working set of a workload is small enough to fit into the 256KB local cache capacity, the L2 miss rate is likely to be small. Thus, the lower L2 hit latency of the private design dominates the performance. This is the case for workloads `BT`, `FT`, `apache`. Second, if the working set is much larger than the total on-chip cache capacity, even the shared design cannot hold the working set. Thus, both private and shared designs will have

high off-chip miss rate, prompting the private design to have better performance through low L2 hit latency. This is the case for workloads CG, EP, LU.

For all workloads but CG, victim replication and victim migration are able to create a significant number of replica hits to reduce the cross-chip fetch latency at the expense of a small increase in off-chip miss rates. The performance of these techniques are usually within 5% of the private design.

Workload CG has a very high L1 cache miss rate at around 10%, but over 70% of the L1 misses hit in the L2 cache, magnifying the low latency advantage of the private design. Our hybrid techniques significantly outperform the shared design but still fall short of the private design.

Shared Design Better Than Private Design

If the difference in off-chip miss rates is significant, we expect the shared design to win even though it has many fewer local hits because it minimizes expensive off-chip misses. This is the case for workloads MG, SP, dbench, and checkers.

Both victim replication and victim migration create replicas for reduced hit latency (shown by the significant number of replica hits) at the expense of slightly increased off-chip miss rate, and achieve significant improvements over the shared design.

It is possible for an application to have a very large working set, yet with little reuse in its data access patterns. In this case, the number of L2 replica hits created by our techniques is low, and its benefit is outweighed by the additional off-chip accesses introduced by the global block evictions. In this case, our techniques can reduce performance. We did not encounter such an application in our workload suite, but a simple miss rate monitor could perhaps be used to limit the replica creation rate to overcome this problem.

6.1.2 Victim Replication versus Victim Migration

Victim migration works slightly better than victim replication for the multi-threaded workloads by storing the tags of actively shared cache blocks in the VM tag array, vacating some of the actual data storage space. This space is split between additional replicas and unshared global blocks. Having more replicas is likely to increase the number of local L2 hits, and having more global blocks is likely to reduce the off-chip miss rate. Both of the scenarios can be observed by comparing the third and fourth bars in Figure 6-5.

6.1.3 Other Configurations

As we increase the on-chip capacity, whether private or shared design works better changes even for the same application depending on whether its working set fits into the given cache configuration. Figure 6-6 presents a pictorial view of how the four policies work depending on the relationships between the cache sizes and the size of the workload's working set. For

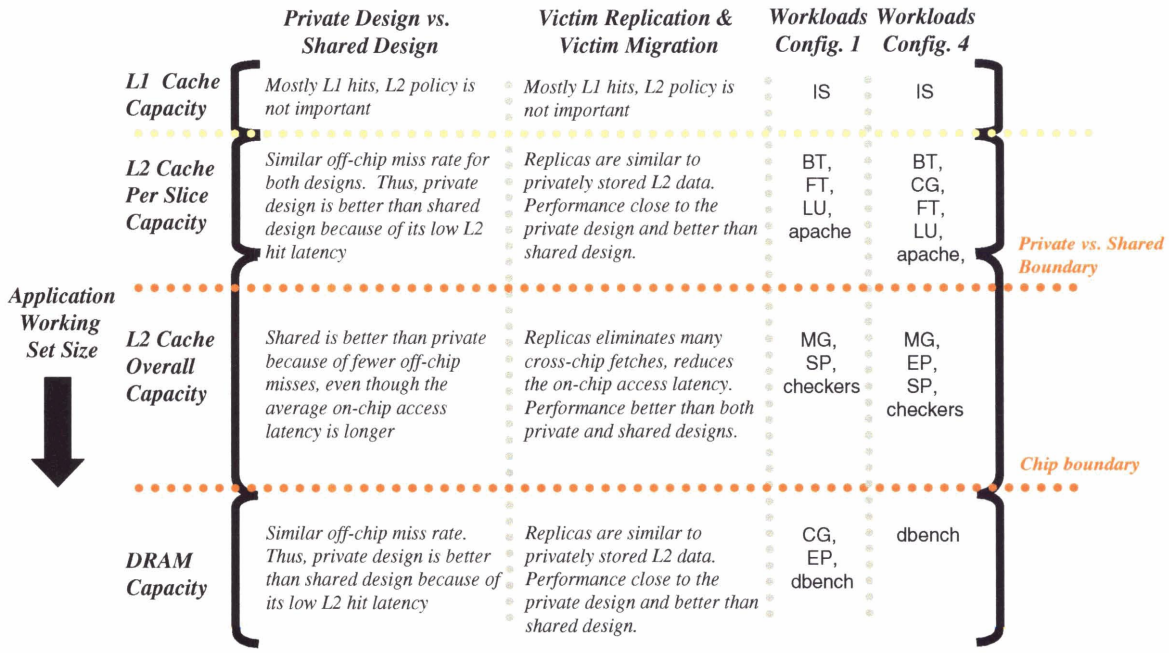


Figure 6-6: Categorization of the behaviors of the different applications according to the relative ratio of the application's working set and the size of the per-slice and overall L2 cache capacity. The behavior of each of the management policies loosely belong to one of the categories shown. As an example, we categorized the multi-threaded workloads for configurations 1 (the smallest cache configuration) and configuration 4 (the largest cache configuration).

example, workload `EP` does better with the shared design in larger L2 caches because more of its working set starts to fit on-chip, significantly reducing the off-chip miss rate compared to the private design. As another example, workload `SP` does better with a private design in larger cache sizes when more of its working set fits into the 1MB local L2 cache slice. Our two techniques manage to be either the best policy or a close second for all of these workloads.

6.1.4 Adaptive Replication Policy

Figure 6-7 plots the percentage of total L2 cache capacity allocated to replicas for the eleven multi-threaded benchmarks in our benchmark suite against execution time. This graph shows two important features of the our hybrid techniques. First, they are adaptive processes that adjust to the execution phases of the program. We can clearly observe execution phases in `CG`, `FT`, and `dbench`. Second, the victim storage capacity offered by our techniques is much larger than any feasible dedicated hardware victim cache. All workloads reached over 25% replicas, in our case equal to a victim cache of over 50 KB.

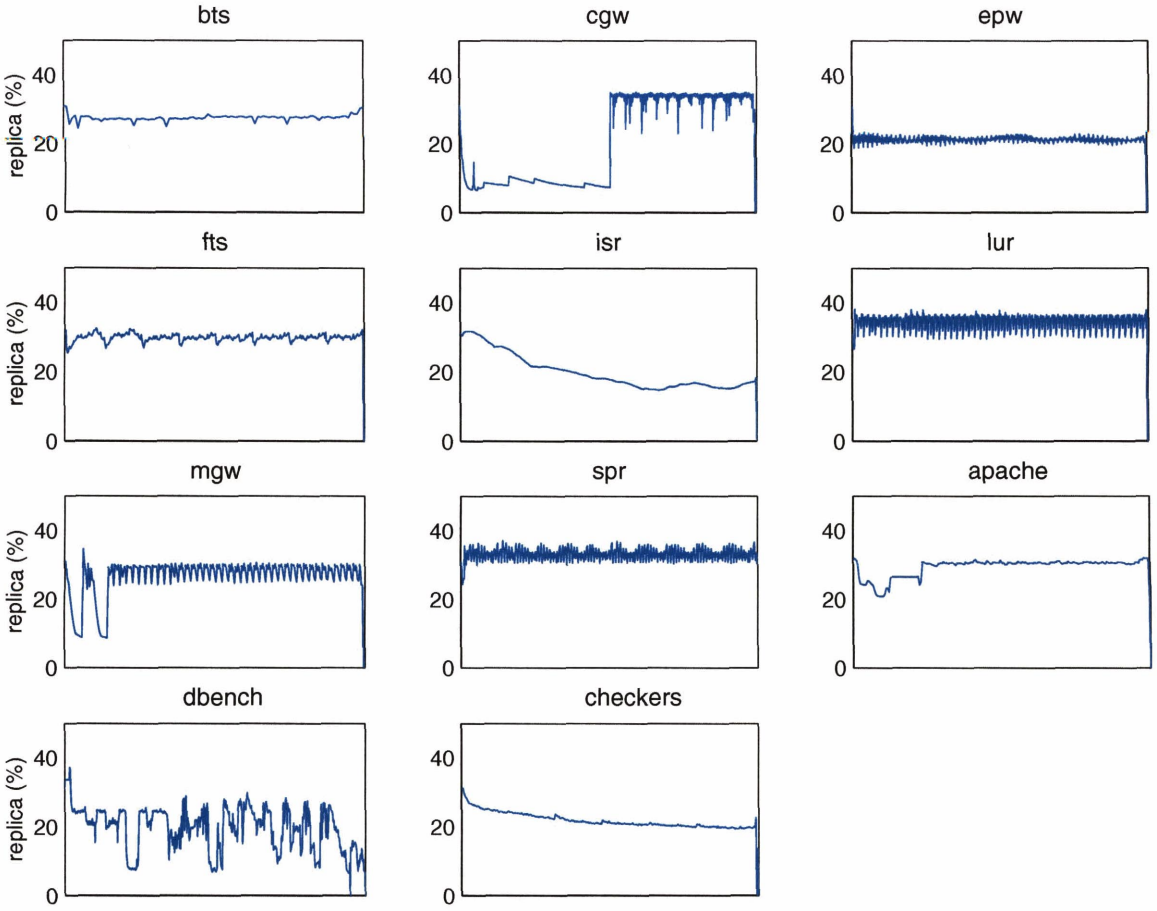


Figure 6-7: Time-varying graph showing the percentage of the L2 allocated to replicas in multi-threaded programs. Average of all eight caches is shown.

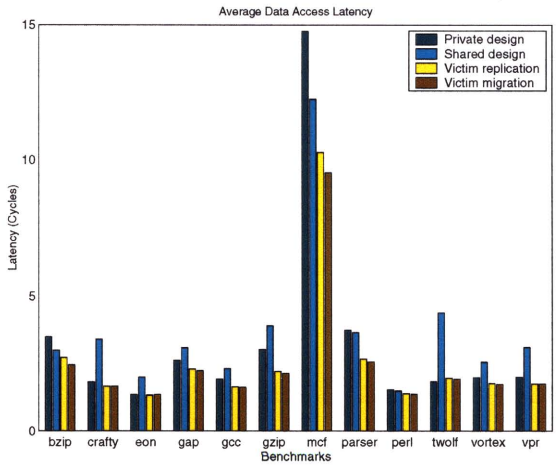


Figure 6-8: Configuration 1: 8KB+8KB/256KB/16FO4. Average access latencies of single-threaded workloads.

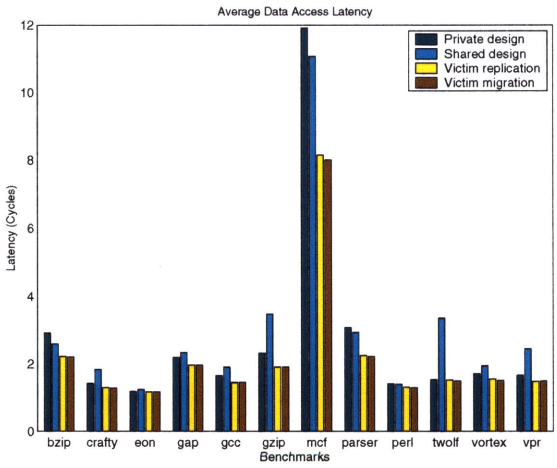


Figure 6-9: Configuration 2: 16KB+16KB/256KB/24FO4. Average access latencies of single-threaded workloads.

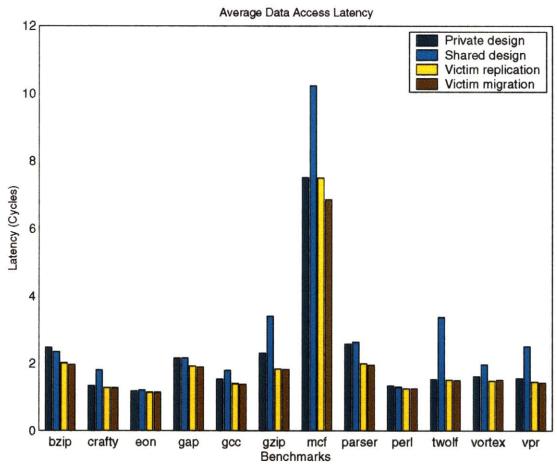


Figure 6-10: Configuration 3: 16KB+16KB/512KB/24FO4. Average access latencies of single-threaded workloads.

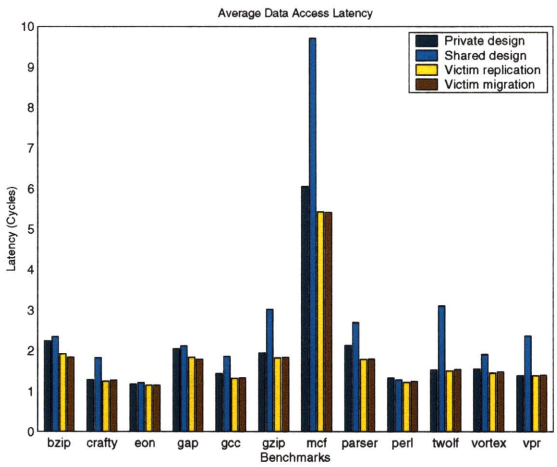


Figure 6-11: Configuration 4: 16KB+16KB/1MB/24FO4. Average access latencies of single-threaded workloads.

Average Access Latency Reduction of Single-Threaded Applications

	Configuration 1 8K+8K/256K/16FO4					Configuration 2 16K+16K/256K/24FO4				
	Reduction (%)					Reduction (%)				
Workload	$\frac{VR}{S}$	$\frac{VR}{P}$	$\frac{VM}{S}$	$\frac{VM}{P}$	$\frac{VM}{VR}$	$\frac{VR}{S}$	$\frac{VR}{P}$	$\frac{VM}{S}$	$\frac{VM}{P}$	$\frac{VM}{VR}$
bzip	9.9	27.9	22.3	42.3	11.2	16.5	30.9	17.5	32.0	0.8
crafty	104.5	9.1	105.0	9.4	0.2	42.6	9.6	43.6	10.4	0.7
eon	50.3	2.2	47.2	0.1	-2.1	6.2	1.6	5.7	1.1	-0.5
gap	34.2	13.7	38.0	16.9	2.8	18.6	11.2	18.9	11.5	0.2
gcc	41.0	17.9	42.6	19.2	1.1	31.8	14.4	31.3	14.0	-0.4
gzip	75.7	36.5	81.6	41.1	3.3	82.5	22.1	81.6	21.5	-0.5
mcf	19.0	43.5	28.5	54.9	7.9	35.6	45.9	38.1	48.6	1.8
parser	36.3	40.2	41.7	45.7	3.9	30.6	36.8	31.8	38.1	0.9
perl	6.7	9.3	9.1	11.8	2.2	6.3	7.5	7.8	9.1	1.4
twolf	123.8	-6.7	127.4	-5.3	1.6	119.4	0.3	123.6	2.3	1.9
vortex	45.0	12.3	48.2	14.8	2.2	25.5	10.4	28.8	13.3	2.6
vpr	77.6	13.9	78.0	14.2	0.2	65.6	12.7	63.8	11.5	-1.1
Avg	52.0	18.3	55.8	22.1	2.9	40.1	17.0	41.0	17.8	0.7
	Configuration 3 16K+16K/512K/24FO4					Configuration 4 16K+16K/1M/24FO4				
	Reduction (%)					Reduction (%)				
Workload	$\frac{VR}{S}$	$\frac{VR}{P}$	$\frac{VM}{S}$	$\frac{VM}{P}$	$\frac{VM}{VR}$	$\frac{VR}{S}$	$\frac{VR}{P}$	$\frac{VM}{S}$	$\frac{VM}{P}$	$\frac{VM}{VR}$
bzip	16.6	23.0	19.4	26.0	2.4	21.6	16.3	27.6	22.1	4.9
crafty	41.5	4.8	42.3	5.4	0.5	45.8	2.8	43.4	1.1	-1.7
eon	5.2	3.0	5.1	2.9	-0.1	5.0	2.3	4.5	1.8	-0.5
gap	13.0	12.4	14.5	13.9	1.3	15.0	11.1	18.0	14.0	2.6
gcc	28.1	9.3	29.8	10.8	1.3	40.7	8.9	39.5	8.0	-0.9
gzip	85.2	25.3	86.2	26.0	0.5	65.6	6.2	65.0	5.8	-0.4
mcf	36.5	0.2	49.2	9.6	9.3	78.6	11.3	79.2	11.7	0.3
parser	31.8	28.6	34.7	31.5	2.2	50.7	19.0	50.1	18.6	-0.4
perl	3.7	6.3	4.2	6.8	0.4	4.7	8.6	3.1	6.9	-1.6
twolf	123.0	1.0	124.2	1.6	0.5	106.0	1.3	101.9	-0.7	-2.0
vortex	32.8	8.6	30.7	6.9	-1.6	31.8	7.0	29.5	5.1	-1.8
vpr	73.3	7.6	75.6	9.1	1.3	71.0	0.3	69.2	-0.8	-1.1
Avg	40.9	10.9	43.0	12.5	1.5	44.7	7.9	44.3	7.8	-0.2

Table 6.2: Average access latency reduction of single-threaded workloads achieved by victim replication and victim migration over the shared and private baseline designs. The five numbers for each workload indicate the percentage reduction of VR to shared, VR to private, VM to shared, VM to private, and VM to VR.

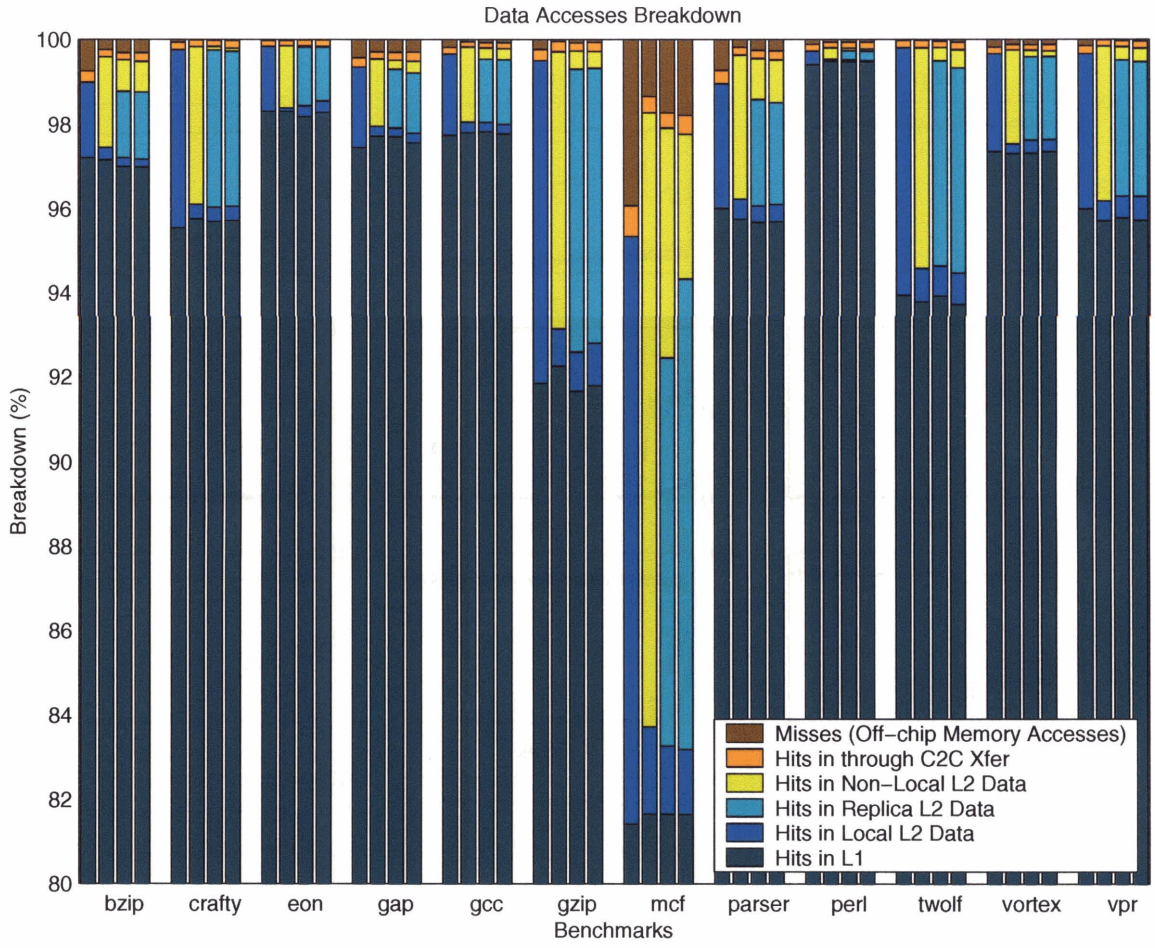


Figure 6-12: Memory access breakdown of single-threaded workloads. Moving from left to right, the four bars for each workload are for the private design, the shared design, victim replication, and victim migration, respectively. Hits are categorized into (from bottom to top): (1) L1 hits; (2) L2 local hits; (3) replica hits; (4) L2 remote hits; (5) cache-to-cache hits; (6) off-chip accesses.

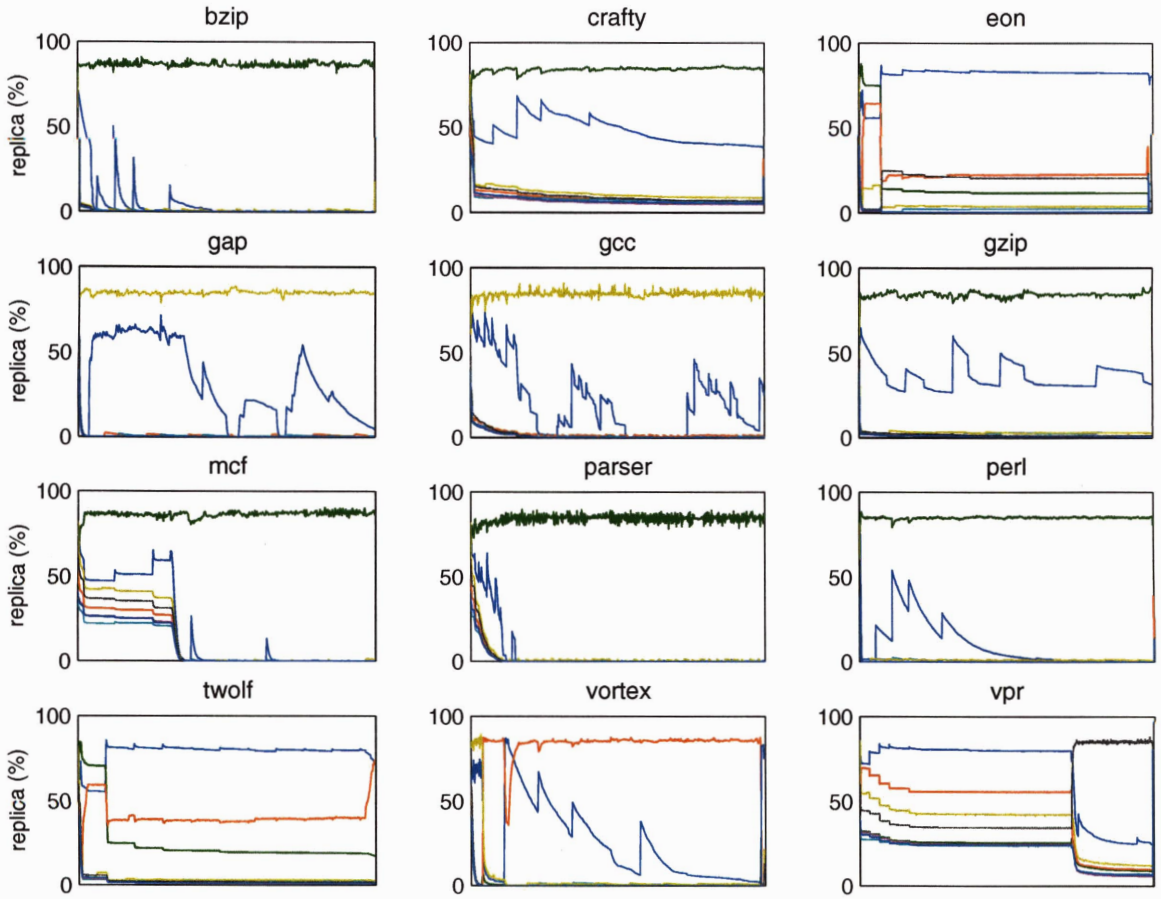


Figure 6-13: Time-varying graph showing the percentage of the L2 allocated to replicas in single-threaded programs. The percentage of replicas in each individual cache is shown.

6.2 Single-Threaded Workloads

We present the same set of results for single-threaded workloads. Figures 6-8 to 6-11 show the average access latency for the single-threaded workloads. Figure 6-12 shows the access breakdown of the four cache designs. Figure 6-13 shows the replica percentage of each individual cache during the execution of victim replication.

6.2.1 Performance Analysis

Figure 6-8 shows the memory access latencies for the single-threaded workloads using configuration 1. In most cases, the shared design performs significantly worse than the other schemes because the L2 hit latency is a critical component of performance for these codes. Table 6.2 summarizes the savings achieved by the victim replication and victim migration over the private and shared baselines. Compared to the shared design, nine out of the twelve workloads achieved 15% or more savings, with six of them over 25%, and an average of 24%.

Victim replication and victim migration are better than the private design for all of the twelve workloads. In several cases, i.e., `bzip`, `gcc`, `gzip`, `mcf`, `parser`, our techniques significantly outperforms both baselines. The performance gain came from two aspects as shown in Figure 6-12. First, victim replication and migration techniques are based on the shared designs, thus have fewer off-chip misses than the private design. Second, they create almost as many local L2 hits through the replicas as the private design, reducing on-chip access latency.

6.2.2 Three-Level Caching

Our hybrid techniques dynamically adapt to a single thread by forming a three-level cache hierarchy: the L1 cache, the local L2 slice, and the remote L2 slices. The local L2 slice can be viewed as the "level 1.5" cache because they hold mostly replicas for the active thread running on the local tile.

This behavior is confirmed by Figure 6-13, which plots the time-varying graph of the percentage of replicas in each of the eight individual L2 caches. For all of these workloads, we observe that one cache holds a very high percentage of replica blocks, usually over 80%. This is the L2 cache of the tile on which the active thread is running. Because we perform full-system simulations and do not attempt to optimize the kernel scheduler to pin the thread on one tile during each run, we sometimes observe the single thread moving between tiles under the control of the scheduler. The replicas "moved" from the old tile to the new one following the thread in benchmarks `eon`, `twolf`, `vortex`, and `vpr`.

Victim Replication versus Victim Migration

Because the single-threaded benchmarks have working sets that are generally smaller than even the smallest configuration simulated (2MB), victim migration did not provide noticeable improvement over victim replication. However, victim migration is either the best policy or a very close second across all benchmarks.

6.3 Multi-Programmed Workloads

Multi-programmed workloads tend to have very little sharing among the different threads, thus the private design is likely to do significantly better than the shared design. Figures 6-14 to 6-17 confirm this intuition, where the shared design is always the worst by a large margin.

6.3.1 Performance Analysis

The performance of victim replication is close to the private design, usually within 5%. Figure 6-18 shows that victim replication can produce significantly more local L2 hits than the shared design with a slight increase in off-chip miss rate.

However, victim replication is not quite as good as the private design. Because there is very little sharing among threads in a multi-programmed workload, each home block is generally used by only one tile, meaning that the cache block is stored twice on the chip (once by the user tile, once by the home tile). This duplication significantly reduces the effective capacity, making victim replication unlikely to win over the private design. This effect is better demonstrated in the smaller cache sizes, where capacity is at a higher premium.

Victim Replication versus Victim Migration

Compared to victim replication, victim migration eliminates the need to keep a duplicate copy at the home tile, behaving just like the private design when necessary. In addition, victim migration allows data to be stored at a global location, stealing limited capacity from other threads when their working sets do not saturate their local L2 slice. This is supported by the lower off-chip miss rate victim migration has over victim replication, shown in Figure 6-18. Overall, victim migration is the best policy for almost all workloads.

While more flexible capacity stealing techniques have been proposed in [CPV05, SSZR05], they are based on snooping coherence protocols that can locate an on-chip cache block relatively easily. Such global searches are complex and can take significant power to achieve in a scalable directory-based design.

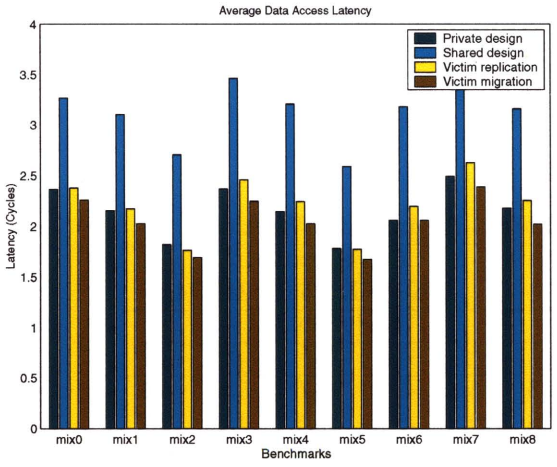


Figure 6-14: Configuration 1: 8KB+8KB/256KB/16FO4. Average access latencies of multi-programmed workloads.

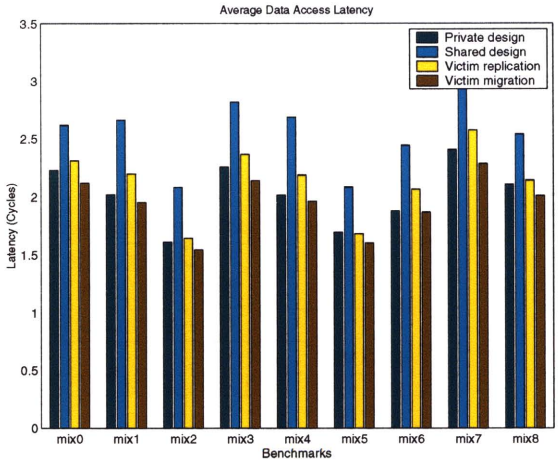


Figure 6-15: Configuration 2: 16KB+16KB/256KB/24FO4. Average access latencies of multi-programmed workloads.

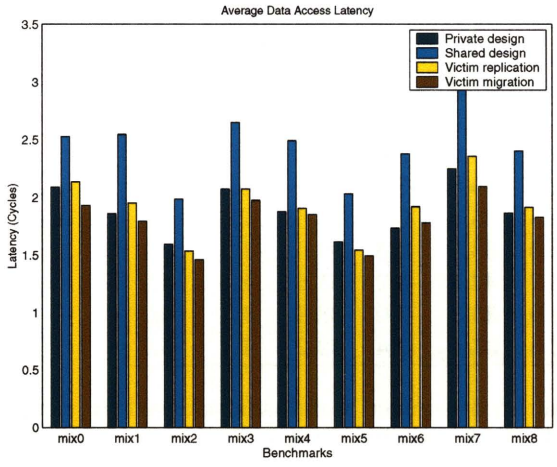


Figure 6-16: Configuration 3: 16KB+16KB/512KB/24FO4. Average access latencies of multi-programmed workloads.

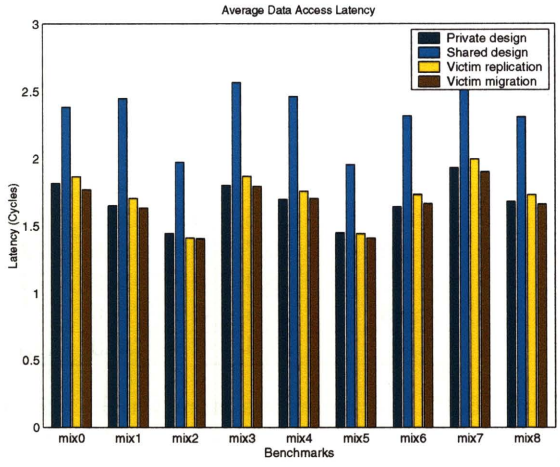


Figure 6-17: Configuration 4: 16KB+16KB/1MB/24FO4. Average access latencies of multi-programmed workloads.

Average Access Latency Reduction of Multi-Programmed Workloads

	Configuration 1 8K+8K/256K/16FO4					Configuration 2 16K+16K/256K/24FO4				
	Reduction (%)					Reduction (%)				
Workload	$\frac{VR}{S}$	$\frac{VR}{P}$	$\frac{VM}{S}$	$\frac{VM}{P}$	$\frac{VM}{VR}$	$\frac{VR}{S}$	$\frac{VR}{P}$	$\frac{VM}{S}$	$\frac{VM}{P}$	$\frac{VM}{VR}$
mix0	37.3	-0.6	44.6	4.7	5.3	13.3	-3.7	23.6	5.1	9.1
mix1	42.7	-0.9	53.1	6.3	7.3	20.9	-8.2	36.3	3.5	12.7
mix2	53.4	3.2	59.7	7.4	4.1	26.6	-2.2	35.1	4.3	6.7
mix3	40.8	-3.7	54.0	5.4	9.4	19.1	-4.6	31.7	5.5	10.6
mix4	42.8	-4.4	58.2	5.9	10.8	22.7	-7.9	37.1	2.9	11.7
mix5	46.1	0.5	54.7	6.4	5.9	23.9	0.8	30.0	5.7	4.9
mix6	44.8	-6.2	54.5	0.1	6.7	18.2	-9.2	30.8	0.5	10.7
mix7	49.0	-5.1	63.9	4.4	10.0	26.2	-6.6	42.2	5.3	12.7
mix8	40.0	-3.4	56.2	7.8	11.6	18.6	-1.7	26.4	4.8	6.6
Avg	44.1	-2.3	55.4	5.4	7.9	21.1	-4.8	32.6	4.2	9.5
	Configuration 3 16K+16K/512K/24FO4					Configuration 4 16K+16K/1M/24FO4				
	Reduction (%)					Reduction (%)				
Workload	$\frac{VR}{S}$	$\frac{VR}{P}$	$\frac{VM}{S}$	$\frac{VM}{P}$	$\frac{VM}{VR}$	$\frac{VR}{S}$	$\frac{VR}{P}$	$\frac{VM}{S}$	$\frac{VM}{P}$	$\frac{VM}{VR}$
mix0	18.4	-2.2	30.9	8.2	10.6	27.6	-2.7	34.6	2.7	5.5
mix1	30.4	-4.9	42.0	3.6	8.9	43.3	-3.2	49.6	1.1	4.4
mix2	29.4	3.7	36.0	9.0	5.1	39.6	2.2	40.3	2.7	0.5
mix3	27.7	0.1	34.0	5.0	4.9	36.9	-3.6	42.8	0.5	4.3
mix4	30.8	-1.6	34.6	1.3	2.9	39.8	-3.5	44.1	-0.5	3.1
mix5	31.6	4.6	36.2	8.3	3.5	35.5	0.3	38.6	2.6	2.3
mix6	23.7	-9.8	33.6	-2.6	8.0	33.5	-5.3	39.0	-1.4	4.1
mix7	28.9	-4.8	45.3	7.3	12.7	44.5	-3.2	51.6	1.5	4.9
mix8	25.5	-2.7	31.6	2.1	4.9	33.3	-3.0	38.9	1.1	4.2
Avg	27.4	-1.9	36.0	4.7	6.8	37.1	-2.4	42.2	1.1	3.7

Table 6.3: Average access latency reduction of multi-programmed workloads achieved by victim replication and victim migration over the shared and private baseline designs. The five numbers for each workload indicate the percentage reduction of VR to shared, VR to private, VM to shared, VM to private, and VM to VR.

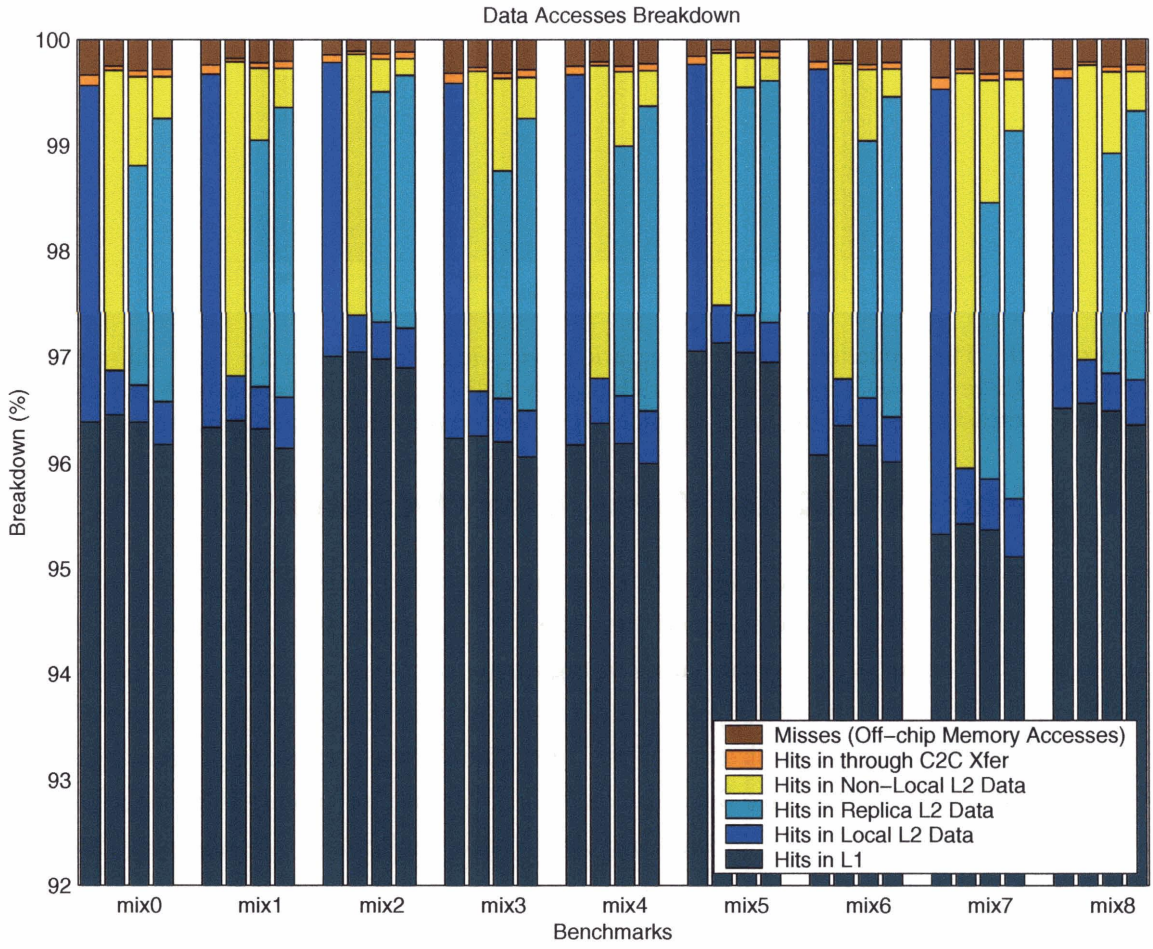


Figure 6-18: Memory access breakdown of multi-programmed workloads. Moving from left to right, the four bars for each workload are for the private design, the shared design, victim replication, and victim migration, respectively. Hits are categorized into (from bottom to top): (1) L1 hits; (2) L2 local hits; (3) replica hits; (4) L2 remote hits; (5) cache-to-cache hits; (6) off-chip accesses.

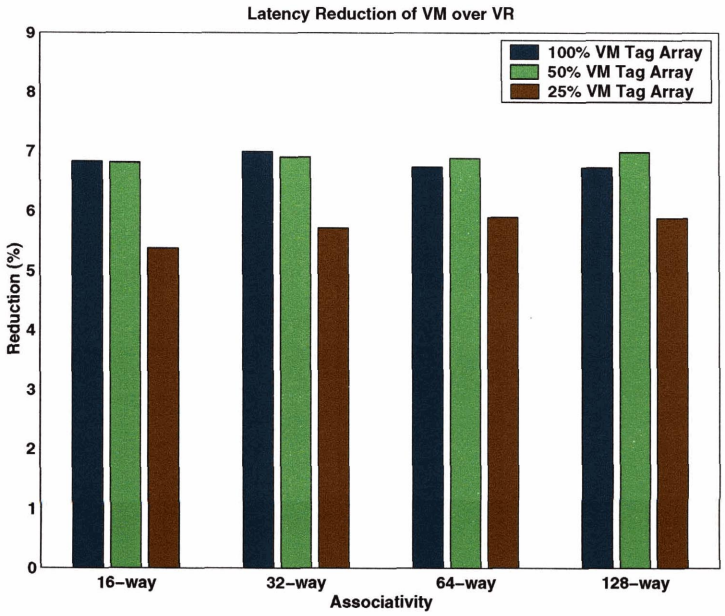


Figure 6-19: The reduction of victim migration over victim replication for three different VM tag sizes. There is little performance degradation by halving the fully-duplicated VM tag array. However, increasing the VM tag array associativity does not provide any performance gain.

6.4 Reducing VM Tag Array Area Overhead

The main drawback of victim migration is the area overhead caused by the VM tag array. For simplicity, we have so far assumed that the VM tag array size is identical to that of the regular L2 tag array. However, the VM tag array can be of any size and associativity. We selected configuration 3 (16K+16K/512K/24FO4) and simulated the performance of two additional VM tag array sizes: at 50%, and 25% of the regular tag array size. The 50% case caused no performance degradation. The 25% case lost about 15% of the latency reduction achieved by the full VM tag array over victim replication. We also experimented with higher VM tag array associativities and observed no noticeable gains.

6.5 Area Comparison of Designs

The vast majority of the area in caches is occupied by data arrays, peripheral circuitry, and interconnects, which is the same for all four designs described in this paper. However, the tag bits, status bits, and in our case, directory bits, all take up non-negligible space. In this section, we provide a simple quantified comparison of the area occupied by the tags and directories for each of these designs. We use the parameters in configuration 1 (8KB+8KB/256KB/16FO4) in the comparison. We also assume a 40-bit physical address width and 64 byte cache block size, which are representative of modern CMP machines [TDJ+02].

Table 6.4 shows the tag and directory area estimates in bits per block used for each

Design Alternative	Tag Width	Directory Entry Width	Total Width	Bit per Block Overhead vs. Shared
Shared	25	9	34	0.0%
Private	28	28	56	4.0%
Victim Replication	28	9	37	0.6%
Victim Migration (1/1)	28	43	71	6.8%
Victim Migration (1/2)	28	26	54	3.7%
Victim Migration (1/4)	28	20	48	2.6%

Table 6.4: Cache area overhead of different designs.

design. It also shows the total bits overhead compared to the shared design, which requires the least area. The actual overall cache area overhead is likely to be much smaller than the ones in Table 6.4 when the area of peripheral circuitry and interconnects are taken into consideration.

In the shared design, the address is used to index a single large shared cache, the width of the tag is smaller than that of the private design. In the eight-tile configuration, three bits are used to select a home tile, making the shared tag 3 bits shorter than that of the private design. The directory uses an 8-bit wide sharing vector. It also leverages the existing valid and dirty status bits to represent state, adding only one extra state bit in our design, for a total of a 9-bit directory. The private design uses the largest area, by having a wider tag and a fully duplicated tag array to maintain the on-chip directory.

For victim replication, the L2 tag must be wide enough to hold physical addresses from any home tile, thus the tag width becomes the same as the private design. Global L2 blocks redundantly set these bits to the address index of the home tile. Replicas of remote blocks can be distinguished from regular L2 blocks as their additional tag bits do not match the local tile index. The full version of victim migration incurs the largest area overhead of all six designs. It consists of all of the components used in victim replication, as well as the VM tag array. However, the overhead can be reduced to less than that of the private design by halving the size of the VM tag array, which gives no significant performance degradation.

6.6 Coherence Traffic Reduction

An additional benefit of the victim replication and victim migration is the reduction of coherence traffic. Compared to the private design, victim replication and victim migration minimizes off-chip traffic, significantly reducing power consumption caused by remote DRAM accesses. Compared to the shared design, victim replication and victim migration eliminate some inter-tile messages when accesses can be resolved in local replicas. Figures 6-20 to 6-22 show the number of coherence messages per thousand instructions executed, weighed by the number of hops each message traversed. While the figures show that the bandwidth of the on-chip switch network is not a bottleneck, reducing the on-chip traffic can dramatically reduce the power consumption of on-chip switch routers. The reduction

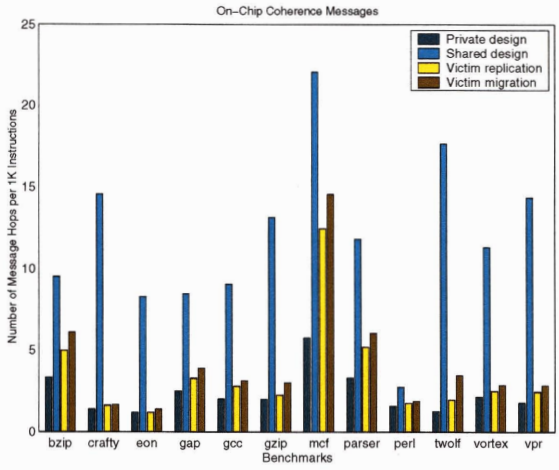


Figure 6-20: On-chip coherence traffic for single-threaded workloads. Traffic is measured in number of messages per hop.

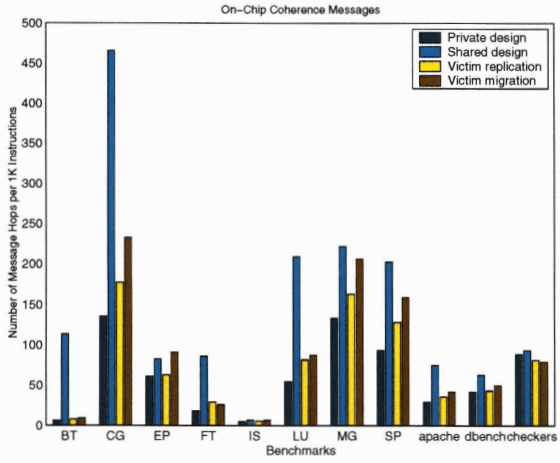


Figure 6-21: On-chip coherence traffic for multi-threaded workloads. Traffic is measured in number of messages per hop.

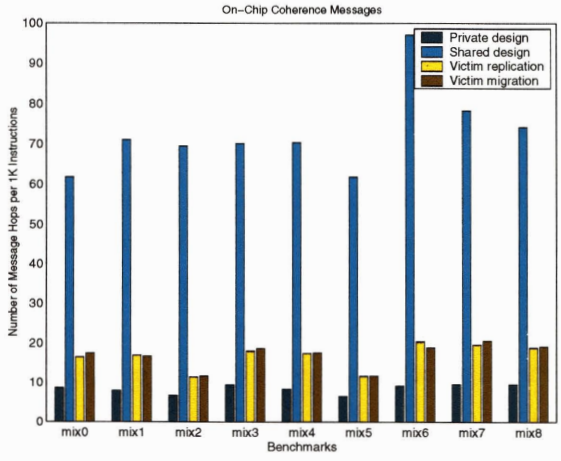


Figure 6-22: On-chip coherence traffic for multi-programmed workloads. Traffic is measured in number of messages per hop.

	Single-Threaded	Multi-Threaded	Multi-Programmed
VR over Private	7.9% — 18.3%	4.0% — 5.8%	-4.8% — -1.9%
VR over Shared	40.1% — 52.0%	8.4% — 18.1%	21.1% — 44.1%
VM over Private	7.8% — 22.1%	7.6% — 8.9%	1.1% — 5.4%
VM over Shared	41.0% — 55.8%	12.2% — 22.4%	32.6% — 55.4%

Table 6.5: Average latency reduction achieved by victim replication and victim migration over the baseline private and shared designs for all three different classes of applications.

achieved by victim replication is usually better than victim migration. This is because for victim migration, a non-negligible percentage of the hits are serviced through a three-way cache-to-cache transfer, causing additional traffic.

6.7 Summary

In Chapter 4, we showed that the implementations of victim replication and victim migration require only simple changes from the baseline shared design. The results presented in this chapter further confirm that victim replication and victim migration are robust, i.e., they work well across single-threaded, multi-threaded, and multi-programmed applications. This can be seen from the brief summary of results in Table 6.5. Finally, the implementations of victim replication and victim migration incur very little area overhead, with a maximum of 3.7% over the baseline shared design, which has the smallest area requirement.

Chapter 7

Conclusions and Future Work

Single-chip multiprocessors have entered the mainstream microprocessor market. Instead of devoting on-chip real-estate to larger data structures and exotic microarchitectural tricks, CMPs achieve higher performance by replicating processor cores and by exploiting thread-level parallelism. Compared to the wide superscalars which have been the driving force of the microprocessor market for the past fifteen years, CMPs directly address several issues that have stalled the continued development of these wide superscalars. First, CMPs can achieve lower energy per operation by utilizing less aggressive cores and still achieve high performance through application parallelization. Second, they can drastically reduce the redesign cycle for each subsequent generation of processors by reusing previous processor designs.

Future CMPs are likely to continue to increase both the number of cores and the total cache capacity on-chip. One key design consideration for these CMPs is how to manage their large cache storage, as the effective data fetch latency heavily impacts the processor performance. In this thesis, we present detailed study of the two baseline cache management policies for these CMPs, private and shared designs, suited for different classes of applications. We introduced two latency reduction techniques, *victim replication* and *victim migration* that can dynamically adjust between the private and the shared cases to optimally place data on-chip, minimizing overall fetch latency.

7.1 Thesis Summary and Contributions

This thesis focuses on tiled CMPs, a class of the CMPs which we believe will become more popular due to its regularity and scalability. The nodes, which we call tiles, are replicated in a two-dimensional mesh. Each tile contains a processor core, L1 caches, a slice of the total L2 capacity on-chip, and a network switch to communicate with the rest of the chip. Cache coherence among all tiles is maintained through a scalable directory-based protocol.

Two major components that govern the fetch latencies in a CMP are the off-chip miss rate and the average on-chip fetch latency. A good cache management policy must consider

both of these conflicting constraints. We examine two baseline L2 cache designs, private design and shared design that demonstrate the trade-offs of these two components. A private design has short on-chip fetch latency, but generally has higher off-chip miss rate than a shared design. A shared design provides the maximum amount of on-chip storage, but on-chip fetches may have to travel across-chip, incurring longer latencies. We presented detailed implementation of both policies under a directory-based protocol.

This thesis proposed two novel latency reduction techniques for tiled CMPs. *Victim replication* is a simple hybrid scheme that combines the advantages of private and shared design. Based on the shared design implementation, victim replication builds a local private victim cache, backing up the local L2 slice, expecting the victims to be used in the near future with reduced latency. A set of cache replacement heuristics is given to determine whether and where to place the victims.

Three different types of workloads are used to evaluate the effectiveness of victim replication. For single-threaded workloads, victim replication works extremely well as it effectively moves all of the recently used data into the local L2 slices of the tile hosting the active thread. Similarly, for multi-programmed workloads, replication moves the working set of each thread to the physical tile hosting that thread.

For multi-threaded workloads, there are two main scenarios. First, if the workload's working set fits within the local cache slice, then the private design will do better than the shared design because it provides short fetch latency. Victim replication also does well by mimicking the behavior of the private design. For workloads with large working sets that do not fit within the local L2 slice, shared design does better than private design because it provides lower off-chip miss rate. Victim replication does better than shared design since it can create replicas with short fetch latency.

We pay extra attention to multi-programmed workloads and introduce *victim migration*, specifically targeting these workloads. Since there is very little sharing among the threads, we remove the need to keep the actual data block at the home tile of the block and simply keep the tag and directory information. The space freed up by victim migration can be used to store more useful data, reducing the off-chip miss rate. A set of replacement policies is presented to complement the operations of victim migration. For multi-programmed workloads, victim migration is able to achieve better performance than all the other three policies discussed in this thesis.

We used a full-system x86 emulator running Linux 2.4.24 as our processor model to drive a detailed cache and memory simulator that implements the various management policies. Experimental results show that for the four typical configurations simulated, our replication techniques outperform most of the 32 applications used in this thesis.

Victim replication and victim migration are much simpler to implement, more flexible, and more scalable than any other proposed related work in reducing cache access latencies. In addition, by using a directory-based protocol, we remove the need of a global snoopy bus

for large node counts [SSZR05, CPV05, HKS+05].

In doing so, we indeed sacrifice some flexibility by having to statically map each data block to a fixed home tile, but avoid the global associative smart search required by all other work. Our approach results in two *simple*, *scalable*, and *robust* cache latency reduction techniques.

Before concluding this thesis, we point out some of the limitations of our experimental infrastructure that still need to be examined further to better understand the effectiveness of the proposed techniques. We conclude our discussion by presenting some possible future work this thesis can lead to.

7.2 Simulation Infrastructure Limitations

In the initial phase of this thesis research, we examined several choices of multiprocessor simulators to use, including Bochs [Law], Simics [MCE+02], other proprietary simulators, and an in-house custom simulator. Due to the limited time frame of the project and the availability of the tools, we chose to use Bochs, an x86 emulator. The full-system nature of the Boch simulator led us to observe the effect of the operating system, and it is open-source so that we were able to easily integrate it with the detailed cache and memory simulator.

However, as mentioned in Chapter 5, a more accurate processor simulator is necessary to further evaluate the effectiveness of our techniques. Bochs is merely an emulator that does not simulate any architectural features of the processor. Specifically, features outlined in Chapter 2, such as prefetching and multi-threading, can help with latency hiding in CMPs.

We were also limited in the number of processors (eight) we can simulate on our Linux port and the Bochs simulator. We anticipate the performance improvement obtained by our hybrid techniques will be more significant at higher core counts because the cross-chip latencies in larger chips will be higher.

7.3 Future Work

In this thesis, we discussed some of the fundamental issues in designing an efficient cache and memory hierarchy for future CMP systems and proposed some solutions. However, as CMPs are a new and fast-evolving architectural target, many challenges lie ahead. In this section, we outline some of these challenges and present our views on how to approach them.

Future CMPs will have higher core counts and larger on-chip caches. If we maintain the even data distribution across the L2 cache slices, the average distance between the requestor and the home tile will also grow accordingly. Even though victim replication and victim migration can create local copies of the shared data, they cannot reduce the latencies of the initial trip to fetch the data from the home tile, as well the inquiries to the directory entries

at the home tile thereafter. Thus, we examine some possibilities of altering the on-chip data mapping to minimize these two factors in fetch latency.

7.3.1 Using Hierarchy

Figure 7-1 shows an approach to reduce the long directory access latency using hierarchy. We show a 16×16, 256-core tiled CMP, a product of continued technology scaling. If we evenly distribute data and/or directories across the entire chip, accesses to data and directory entries will be increasingly more expensive because they incur cross-chip communications.

The goal in using hierarchy is to have the majority of the accesses to directories be handled by a *local directory*, which is located in a nearby tile and much faster to get to than the actual global directory.

Using Regions

To minimize these cross-chip communications, we divide the tiled CMP into *coherent regions*. For example, regions *R1*, *R2*, and *R3* in Figure ?? are all 2×2 coherent regions. Such a *coherence region* operates as an independent tiled CMP with respect to the rest of the chip, and maintains its own coherence. Any latency reduction techniques can be used within a coherence region, and data coherence between multiple regions are kept at the home tile of the actual data. Each tile in this case would carry a directory to maintain coherence within the region, as well as a directory to maintain coherence across all regions. The example in Figure 7-1 shows the sharing of a data block whose global home is mapped to the upper-left tile of region *R3*. Each individual coherence region must also cache a local home directory entry to maintain data coherence among all the tiles inside the region. This example shows a shared design for each region, and duplicated tag directory to keep all the regions coherent. Specific implementations can choose to use any cache management policies within each coherent region and among all regions. Furthermore, it is also possible to allow a region to be incoherent.

Partition Algorithms

A challenge in the hierarchical approach is to find the appropriate partitions for these regions to gain the optimal performance. One approach is to use profiling information and statically partition the CMP array [HKS+05]. A more appealing approach is to leverage the operating system to help determine the optimal partitioning dynamically.

7.3.2 Leveraging Software

The operating system can give us valuable hints in the sharing patterns of workloads, as illustrated in Figure 7-2. (We again use a 16×16, 256-core tiled CMP as our target architecture.) The idea here is to use the operating system to allocate the threads to

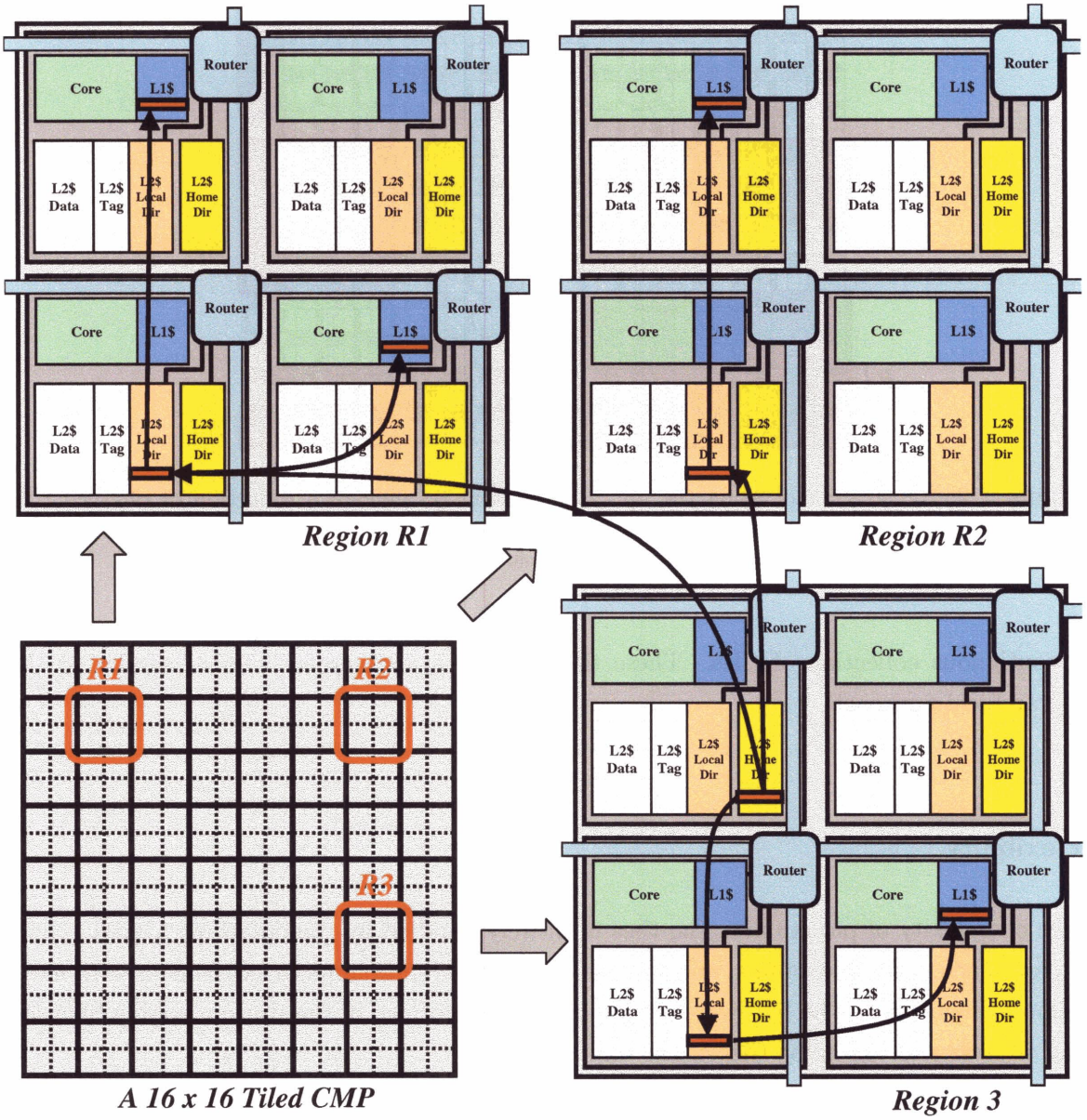


Figure 7-1: Illustration of hierarchical cache coherence for CMPs. In this example, each 2×2 square forms its own coherence region and the cache storage located within the region is shared by all processor cores within the region. However, when two regions, e.g., regions 1 and 2 share data, there is a directory entry on the home node that keeps track of all the data for each region.

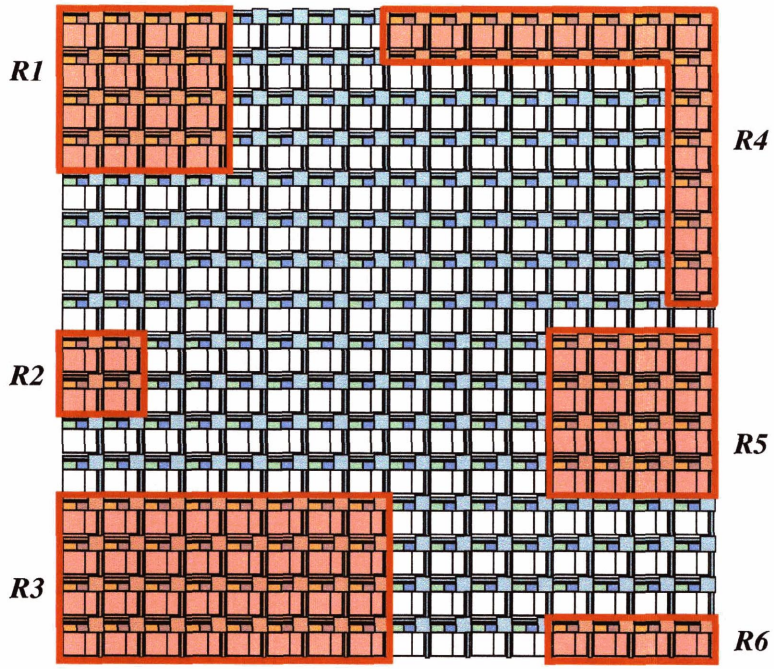


Figure 7-2: Illustration of using the operating system to allocate a collection of physical tiles for each independent program running on the CMP. The operating system is fully responsible for maintaining cache coherence within different regions.

a contiguous collection of tiles. These tiles will form a coherence region, similar to the hierarchical approach.

Flexibility

The complexity of partitioning the tiles is entirely handled by the operating system. Such an approach is flexible, online, and can use operating system hints to experiment with more complex heuristics. The hardware simply has to be informed of the static mapping between the address and the home tiles of the data blocks.

Because partitioning is dynamically adjustable to suit the usage of the workload, the coherent regions can be of different sizes and shapes to optimally accommodate the characteristics of the program.

One main drawback of the software approach is that the cache content is likely to be flushed, depending on how the operating system partitions the regions and whether the static mapping needs to change. This requirement, however, is unlikely to cause major performance degradation for multi-programmed workloads. An additional problem here could be region fragmentation, because the operating system needs to partition and rejoin different coherent regions at various times.

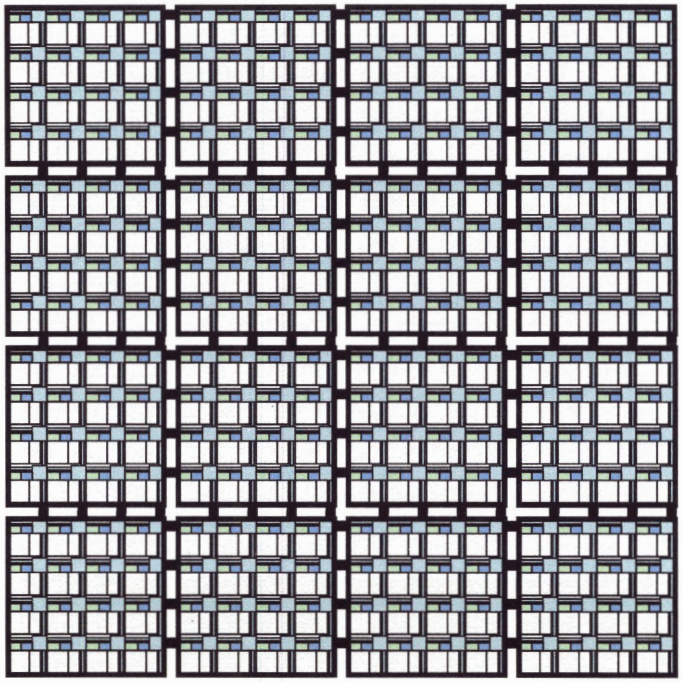


Figure 7-3: Illustration of using multiple moderate-sized tiled CMPs to form a massive many-core CMP system. The integration between neighboring chips is tight.

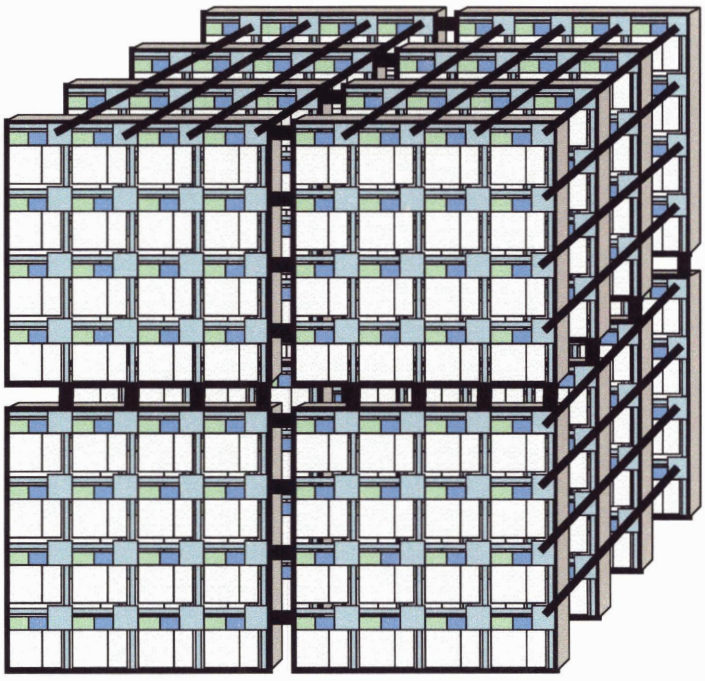


Figure 7-4: Illustration of forming a multi-chip CMP system in three-dimensional fashion.

7.3.3 Future CMP Topology

Figure 7-3 shows an example of such a system, which consists of many moderately sized tiled CMPs to form a massively parallel machine. Furthermore, various new silicon emerging technologies could allow multiple dies to be connected together in a three-dimensional fashion, forming a tiled CMP cube as shown in Figure 7-4. These newer CMP topologies present different trade-offs in cache and memory latencies. Thus, inventing flexible latency-reduction techniques to manage the cache and memory of these new architectures will be both challenging and vital to the performance of these machines.

Appendix A

Cache Coherence Protocol Implementation

In this chapter, we briefly present the basic aspects of the cache coherence protocol used in our memory system. A typical protocol can be described through three separate components: 1) the *coherence states* associated with each cache or memory block, 2) the different types of *coherence messages* communicated between the tiles, 3) the *coherence actions* taken by the coherence controllers upon receiving the processor request, the coherence messages, and the off-chip DRAM messages. Such actions may include block state transitions or generating reply messages. In the following, we present each of these three components in our protocol.

A.1 Coherence States

This section presents the coherence states for the L1 cache, the L2 cache, and the DRAM.

A.1.1 Memory Block States

The simplest module to implement in our coherence protocol is the physical memory (DRAM). Traditionally, directories are stored in the DRAM. However, as we discussed in Chapter 3, we implement a *perfect* on-chip directory cache for all cached data blocks on-chip by duplicating tags. By doing so, we have removed the need to implement off-chip directories, as shown in Figure A-1. When a block that is not on-chip is first requested by any processor core, the request is propagated to the off-chip DRAM. The DRAM simply returns the data to the home tile of the requested block. A directory entry is created once the DRAM reply carrying the requested data reaches the home tile. The directory entry is either in the true directory format for shared designs, or in the duplicated tag array format for the private design. When the last copy of a cached block is evicted from the chip, the on-chip directory is cleared and any dirty data is written back to the off-chip memory.

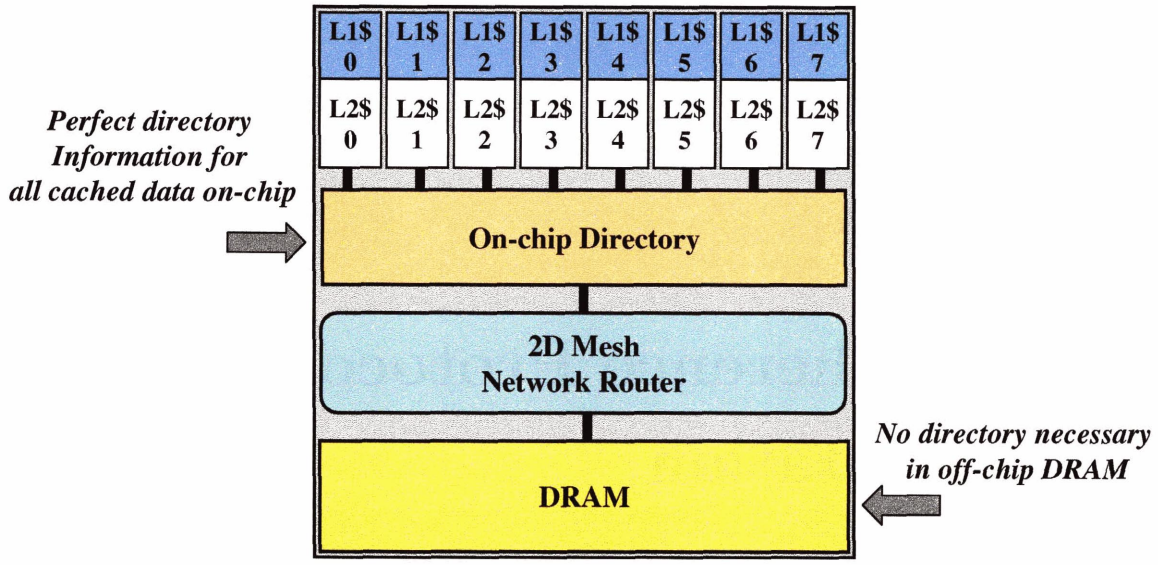


Figure A-1: Implementing a perfect directory for all cached data on-chip removes the need to have directories in the off-chip DRAM. The on-chip directory cache is guaranteed to have all the necessary sharing information of any cached block.

A.1.2 L1 Cache Block States

The L1 cache's states are list in Table A.1. There are four stable MESI states, and one transient state indicating that there is an outstanding request being serviced.

A.1.3 L2 Cache Block States

The L2 cache's coherence states are list in Table A.2. The states include the four stable MESI states, as well as two transient states. The two transient states are reached during cache reply-forwarding and hold the final stable state to enter once the transaction is complete. In addition to holding the state, each L2 block is associated with a presence vector to keep track of all the sharers. A full presence vector is used in our system as the number of tile is small with 1 bit per tile.

A.2 Coherence Messages

Table A.3 summarizes all of the coherence message types used by our protocol. The twenty-six messages are prefixed to make them easier to read. We use **c** (cache) to indicate requestors, sharers, or owner, and **h** (home) to indicate the home tile. We also use **q** to indicate a request message and **p** for a reply message. In addition, an message could carry a payload of one cache block, and we use a suffix **D** to indicate that. The messages are divided into the following five groups: 1) Type *chq*: request messages from requestor to home tile. 2) Type *hcp*: reply messages from home tile to requestor. 3) Type *hcq*: reply-forwarding messages from home tile to owner/sharers. 4) Type *chp*: reply-forwarding

Group	States	Description
Stable	INV	The **I** state, indicating an invalid cache block.
	SHR	The **S** state, indicating a read-only block is cached in this L1 cache and possibly in other L1 caches as well.
	CEX	The **E** state, indicating a clean block is cached in this L1 cache only. No other L1 cache has a copy.
	DEX	The **M** state, indicating a writable (dirty) block is cached in this L1 cache only. No other L1 cache has a copy.
Transient	BSY	Indicating a request is outstanding for this cache block.

Table A.1: Coherent states of the L1 cache blocks include four stable MESI states and one transient state.

Group	States	Description
Stable	INV	The **I** state, indicating an invalid cache block.
	SHR	The **S** state, indicating a read-only block is cached in this L2 cache and possibly in other L2 caches as well.
	CEX	The **E** state, indicating a clean block is cached in this L2 cache only. No other L2 cache has a copy.
	DEX	The **M** state, indicating a writable (dirty) block is cached in this L2 cache only. No other L2 cache has a copy.
Transient	BSH	The busy-shared state, entered when a shared read request is received but cannot be serviced immediately due to a coherence miss. Downgrade request is sent to the owner and a revision block then sent to the requestor. Wait for reply before entering SHR state.
	BEX	The busy-exclusive state, entered when an exclusive read request is received but cannot be serviced immediately due to a coherence miss. Invalidation request(s) are sent to owner/sharers and a revision block then sent to the requestor. Wait for reply before entering EXC state.

Table A.2: Coherent states of the L2 cache blocks include four stable MESI states and two transient states.

Message Group	Message Type	Message Description
Cache→Home Request	chqRSH	Shared read request.
	chqREX	Exclusive read request.
	chqWBKD	Writeback request.
	chqDRP	Explicit drop request.
Home→Cache Reply	hcpRSHD	Shared read reply.
	hcpREXD	Exclusive read reply.
	hcpUPG	Upgrade reply.
	hcpRUAD	Exclusive read reply.
	hcpREV	Exclusive read revision.
	hcpREVD	Exclusive read revision with data.
	hcpWBK	Writeback acknowledgment.
	hcpDRP	Explicit drop acknowledgment.
	hcpNAK	Negative acknowledgment.
Home→Cache Request	hcqINV	Invalidation intervention request.
	hcqDNG	Downgrade intervention request.
	hcqCCX	Cache-to-cache transfer intervention request.
Cache→Home Reply	chpINV	Invalidation reply from a clean-exclusive block.
	chpINVD	Invalidation reply from a dirty-exclusive block.
	chpDNG	Downgrade reply from a clean-exclusive block.
	chpDNGD	Downgrade reply from a dirty-exclusive block.
	chpREV	Revision reply.
	chpREVD	Revision reply with data.
Cache→Cache Transfer	ccpINV	Invalidation reply from a clean-exclusive block.
	ccpINVD	Invalidation reply from a dirty-exclusive block.
	ccpDNGD	Downgrade reply from a dirty-exclusive block.
	ccpCCXD	Cache-to-cache reply from a shared block.

Table A.3: The types of coherence messages used in this protocol. The first two letters of the prefix signifies whether the message is from the sharing cache to the home tile (ch), home tile to the sharing cache (hc), or cache-to-cache transfers (cc). The third letter of the prefix indicates whether the message is a request message (q) or a reply message (p). Messages that end in D carry a payload.

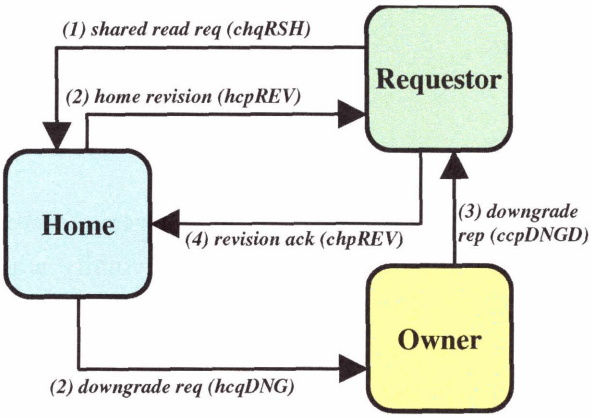
messages from owner/sharers to the home tile. 5) Type *ccp*: cache-to-cache reply messages from owner/sharers to the requestor.

A.3 Coherence Actions

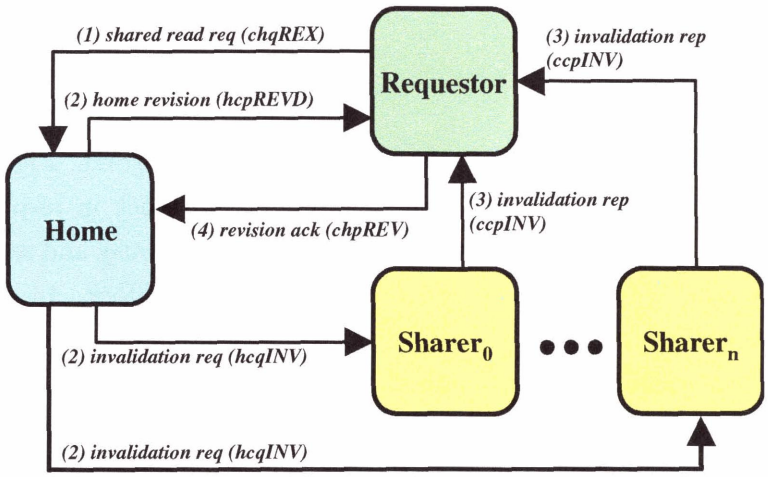
This section describes the coherence actions taken by the coherence controllers co-located with the L1 cache and the L2 cache. The DRAM behaves normally without the burden of maintaining coherence. The actions are summarized in Table A.4 on page 115 and Table A.5 on page 116. To simplify our discussions, these actions summarized in these two tables only represent the main portions of the protocol and ignores some of the cumbersome implementation details and corner cases. Further, to complement and to facilitate the understanding of the coherence action tables, we also show several examples in Figure A-2 on page 114.

A.3.1 Examples

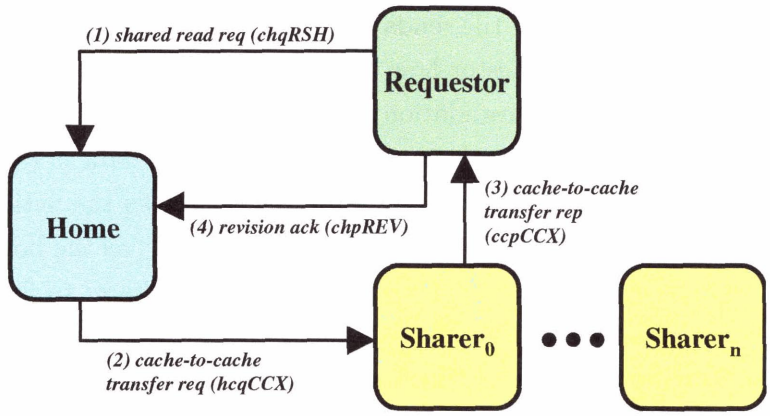
Figure A-2 shows four examples of how the coherence protocol works. Figure A-2(a) shows the reply-forwarding action sequence of an exclusively held block in response to a shared read request. Upon receiving the request, home sets its state to busy and sends a downgrade request with the tile ID of the requestor to the owner of the block. In addition, a home revision reply is sent to the requestor, telling it to expect one downgrade message. Once the requestor receives the downgrade message, it considers the request complete and refills its cache. It must also notify the home tile that the data is received by sending home an acknowledgment. Upon receiving this acknowledgment, home tile completes the request and moves into the shared stable state. Figure A-2(b) shows the action sequence of a shared block in response to a exclusive request, which is similar to the previous one. Instead of sending a downgrade request, home tile sends an invalidation request to each of the sharers. The revision message tells the requestor how many invalidation replies to anticipate. Once the requestor receives all of the invalidation messages, it considers the request complete and refills its cache. It must also notify the home tile that the data is received and home subsequently completes the request. Lastly, Figure A-2(c) shows the action sequence of a shared block in response to a shared request. This case is simple, as the home tile chooses a sharer and sends a cache-to-cache transfer request, asking the sharer to forward the actual data to the requestor. However, this case only happens in the private design and the victim migration design, in which cases the home tile may hold only the directory entry, but not the actual data.



(a) Shared read through downgrading current owner



(b) Exclusive read through invalidating current sharers



(c) Shared read through cache-to-cache transfer from a sharer

Figure A-2: Examples of reply-forwarding used in the coherence protocol. Figure (a) shows the action sequence of an exclusively held block in response to a shared read request. Figure (b) shows the action sequence of a shared block in response to a exclusive request. Figure (c) shows the action sequence of a shared block in response to a shared request.

Request or Message	Initial State	Final State	Output Message	Cache Action and Description
Load Request	INV	N/A	chqRSH	L1 miss. Push request into miss buffer.
	SHR	SHR	N/A	L1 hit.
	CEX	CEX	N/A	L1 hit.
	DEX	DEX	N/A	L1 hit.
	BSY	BSY	chqRSH	L1 miss. Push request into miss buffer and merge with preceding requests to the same block if appropriate.
Store Request	INV	N/A	chqREX	L1 miss. Push request into miss buffer.
	SHR	BSY	chqREX	L1 coherence miss. Push request into miss buffer.
	CEX	DEX	N/A	L1 hit.
	DEX	DEX	N/A	L1 hit.
	BSY	BSY	chqREX	L1 miss. Push request into miss buffer and merge with preceding requests to the same block if appropriate.
hcqINV	Any	INV	ccpINV[D]	Invalidates block. Dirty block attached if appropriate.
hcqDNG	[C|D]EX	SHR	ccpDNGD	Downgrades block. Data block attached.
hcqCCX	SHR	SHR	ccpCCXD	Sends shared data directly to the requestor.
hcpRSHD	N/A	SHR	N/A	L1 refill.
hcpREXD	N/A	[C|D]EX'	N/A	L1 refill.
hcpUPG	N/A	[C|D]EX'	N/A	L1 refill.
hcpRUAD	N/A	[C|D]EX'	chpREVD	L1 refill.
hcpREV[D]	N/A	CEX'* | SHR'*	chpREV	If all downgrade/invalidation replies have been received, then refill L1. Otherwise, continue waiting.
hcpWBK	N/A	N/A	N/A	Completes the writeback request.
hcpDRP	N/A	N/A	N/A	Completes the explicit request.
hcpNAK	N/A	N/A	Original Request	Reissue the original request. May merge with subsequent requests to the same block if appropriate.
ccpINV[D]	N/A	DEX*	chpREV[D]*	If all other invalidation replies and the home revision (hcpREVD) have been received, then refill L1. Otherwise, continue waiting.
ccpDNGD	N/A	SHR*	chpREV*	If home revision (hcpREVD) has been received, then refill L1. Otherwise, continue waiting.
ccpCCXD	N/A	SHR	N/A	L1 refill.

Table A.4: L1 cache controller actions to processor requests and incoming coherence messages. A (') indicates that one of the multiple states listed will be entered depending on the original request (shared or exclusive). A asterisk (*) means that the state is only entered upon described conditions.

L1/DRAM Messages	Initial State	Final State	Output Message	Cache Action and Description
Any cache request	B[SH|EX]	B[SH|EX]	hcpNAK	Reply with negative acknowledgment.
chqRSH	INV	N/A	To DRAM	L2 miss. Issues request to off-chip DRAM. Push request into miss buffer.
	SHR	SHR	hcpRSHD	L2 hit.
	[C|D]EX	BSH	hcqDNG & hcpREV	L2 coherence miss. Send downgrade request to owner, and sends revision to the requestor.
chqREX	INV	N/A	To DRAM	L2 miss. Issues request to off-chip DRAM. Push request into miss buffer.
	SHR	BEX	hcqINV & hcpREVD	L2 coherence miss. Send invalidation request(s) to all sharers, and sends revision to the requestor.
	[C|D]EX	DEX	hcpREXD	L2 hit. Private design only.
chpINV[D]	BEX	INV	chpINV[D]	In private design only. L1 invalidation reply to the local L2 cache.
chpDNG[D]	BSH	SHR	chpDNG[D]	In private design only. L1 downgrade reply to the local L2 cache.
chpREV[D]	BSH	SHR	N/A	Concludes the shared read request.
	BEX	DEX	N/A	Concludes the exclusive read request.
DRAM reply	BSH	SHR	hcpRSHD	Concludes the shared read request.
	BEX	DEX	hcpREXD	Concludes the exclusive read request.

Table A.5: L2 cache controller actions to L1 requests and DRAM replies.

Bibliography

[AB86] J. Archibald and J. Baer. Cache coherence protocols: Evaluation using a multiprocessor simulation model. *ACM Transactions on Computer System (TOCS)*, 4(4):273–298, November 1986.

[ACJ+99] A. Agarwal, D. Chaiken, K. Johnson, D. Kranz, J. Kubiatowicz, B. Lim, K. Mackenzie, and D. Yeung. The MIT Alewife machine. *Proceedings of IEEE*, 87(3):430–444, March 1999.

[AGGD01] M. Acacio, J. Gonzalez, J. Garcia, and J. Duato. A new scalable directory architecture for large-scale multiprocessors. In *The 7th International Symposium of Computer Architecture*, Nuevo Leone, Mexico, January 2001.

[AHKB00] V. Agarwal, M. Hrishikesh, S. Keckler, and D. Burger. Clock rate versus IPC: The end of the road for conventional microarchitectures. In *The 27th International Symposium on Computer Architecture*, Vancouver, BC, Canada, May 2000.

[ALKK90] A. Agarwal, B. Lim, D. Kranz, and J. Kubiatowicz. April: A processor architecture for multiprocessing. In *The 27th International Symposium on Computer Architecture*, Seattle, WA, June 1990.

[ASHH88] A. Agarwal, R. Simoni, J. Henessy, and M. Horowitz. An evaluation of directory schemes for cache coherence. In *The 15th International Symposium on Computer Architecture*, Honolulu, HI, May 1988.

[AW03] A. Alameldeen and D. Wood. Addressing workload variability in architectural simulations. In *The 9th International Symposium on High-Performance Computer Architecture*, Anaheim, CA, February 2003.

[BBB+94] D. Bailey, E. Barszcz, J. Barton, D. Browning, R. Carter, L. Dagum, R. Fatoohi, S. Fineberg, P. Frederickson, T. Lasinski, R. Schreiber, H. Simon, V. Venkatakrishnan, and S. Weeratunga. The NAS parallel benchmarks. Technical Report RNR-94-007, NASA, March 1994.

[BC91] J. Bear and T. Chen. An effective on-chip preloading scheme to reduce data access penalty. In *International Conference on Supercomputing*, Albuquerque, NM, 1991.

[BGM+00] L. 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 International Symposium on Computer Architecture*, Vancouver, BC, Canada, May 2000.

[BPZA05] K. Barr, H. Pan, M. Zhang, and K. Asanović. Accelerating multiprocessor simulation with a memory timestamp record. In *International Symposium of Performance Analysis and System Simulation*, Austin, TX, March 2005.

[Bur92] H. Burkhardt. Overview of the KSR1 computer system. Technical Report KST-TR-9202001, Kendall Square Research, 1992.

[BW04] B. Beckmann and D. Wood. Managing wire delay in large chip-multiprocessor caches. In *The 37th International Symposium on Microarchitecture*, Portland, OR, December 2004.

[Cav05] Cavium Networks. OCTEON network service processors, August 2005.

[CF78] L. Censier and P. Feautrier. A solution to coherence problems in multicache systems. *IEEE Transaction on Computer*, C-27(12):1112–1118, December 1978.

[CHM96] T. M. Conte, M. A. Hirsch, and K. N. Menezes. Reducing state loss for effective trace sampling of superscalar processors. In *International Conference on Computer Design*, October 1996.

[CKA91] David Chaiken, J. Kubiatowicz, and A. Agarwal. LimitLESS directories: A scalable cache coherence scheme. In *The 4th International Conference on Architectureal Support for Programming Languages and Operating Systems*, April 1991.

[CKP91] D. Callahan, K. Kennedy, and A. Porterfield. Software prefetching. In *The 4th International Conference on Architectureal Support for Programming Languages and Operating Systems*, New York, NY, April 1991.

[Cor91a] Intel Corporation. Paragon XP/S product overview, 1991.

[Cor91b] Thinking Machines Corporation. The connection machine CM-5 technical summary, October 1991.

[Cor93] CONVEX Computer Corporation. Exemplar architecture manual, 1993.

[Cor00] Standard Performance Evaluation Corp. SpecINT2000, 2000.

[CPV03] Z. Chishti, M. Powell, and T. Vijaykumar. Distance associativity for high-performance energy-efficient non-uniform cache architectures. In *The 36th International Symposium on Microarchitecture*, San Diego, CA, December 2003.

[CPV05] Z. Chishti, M. Powell, and T. Vijaykumar. Optimizing replication, communication, and capacity allocation in CMPs. In *The 32nd International Symposium of Computer Architecture*, Madison, WI, June 2005.

[CR05] M. Cameron and B. Rohit. Montecito: A dual-core, dual-thread Itanium processor. *IEEE Micro*, 25(2):10–20, March/April 2005.

[CSG97] D. Culler, J. Pal Singh, and A. Gupta. *Parallel Computer Architecture: A Hardware/Software Approcah.* Morgan Kuffman Publishers, 1997.

[CYS+93] M. Cekleov, D. Yen, P. Sindhu, J. Frailong, J. Gastinel, M. Splain, J. Price, G. Beck, B. Liencres, F. Cerauskis, C. Coffin, D. Bassett, D. Broniarczyk an dS. Fosth, T. Nguyen, R. Ng, J. Hoel, D. Curry, L. Yuan, R. Lee, A. Kwok, A. Singhal, C. Cheng, G. Dykema, S. York, B. Gunning, B. Jacksno, A. Kasuya, D. Angelico, M. Levitt, M. Mothashemi, D. Lemenski, L. Bland, and T. Pham. SPARCcenter 2000: Multiprocessing for the 90's! In *Compcon Spring '93, Digest of Papers*, pages 345–353, February 1993.

[DT99] F. Dahlgren and J. Torrellas. Cache-only memory architectures. *IEEE Transaction on Computer*, 32(6):72–79, June 1999.

[Goo83] J. Goodman. Using cache memory to reduce processor memory traffic. In *The 10th International Symposium on Computer Architecture*, Stockholm, Sweden, June 1983.

[Gro01] MIT Supercomputing Technologies Group. Cilk 5.3.2. http://supertech.lcs.mit.edu/cilk, November 2001.

[Gus92] D. Gustavson. The scalable coherent interface and related standards projects. *IEEE Micro*, 12(1):10–22, February 1992.

[GW94] M. Galles and E. Williams. Performance optimization, implementation, and verification of the SGI Challenge multiprocessor. In *27th Hawaii International Conference on System Science*, volume 1, January 1994.

[HBJ+02] M. Hrishikesh, D. Burger, N. Jouppi, S. Keckler, K. Farkas, and P. Shivakumar. The optimal logic depth per pipeline stage is 6 to 8 FO4 inverter delays. In *The 29th International Symposium of Computer Architecture*, Anchorage, AK, May 2002.

[HKS+05] J. Huh, C. Kim, H. Shafi, L. Zhang, D. Burger, and S. Keckler. A NUCA substrate for flexible CMP cache sharing. In *ICS05*, Cambridge, MA, June 2005.

[HLH92] E. Hagersten, A. Landin, and S. Haridi. DDM — A cache-only memory architecture. 25(9):44–55, September 1992.

[HMH01] R. Ho, K. Mai, and M. Horowitz. The future of wires. *Proceedings of IEEE*, 89(4):490–504, April 2001.

[Hor83] M. Horowitz. *Timing Models for MOS Circuits.* PhD thesis, Stanford University, December 1983.

[HP03] A. Hartstein and T. Puzak. Optimum power/performance pipeline depth. In *The 36th International Symposium on Microarchitecture*, San Diego, CA, December 2003.

[HSU+01] G. Hinton, D. Sager, M. Upton, D. Boggs, D. Carmean, A. Kyker, and P. Roussel. The microarchitecture of the pentium 4 processor. *Intel Technology Journal Q1*, February 2001.

[Inc93] Cray Research Inc. Cray T3D system architecture overview, 1993.

[JLGS90] D. James, A. Laundrie, S. Gjessing, and G. Sohi. Distributed-directory scheme: Scalable coherent interface. *IEEE Transaction on Computer*, 23(6), June 1990.

[KAO05] P. Kongetira, K. Aingaran, and K. Olukotun. Niagara: A 32-way multithreaded Sparc processor. *IEEE Micro*, 25(2):21–29, March/April 2005.

[KBK02] C. Kim, D. Burger, and S. W. Keckler. An adaptive, non-uniform cache structure for wire-delay dominated on-chip caches. In *The 10th Interational Conference on Architectural Support of Programming Languages and Operating Systems*, San Jose, CA, October 2002.

[Kes99] R. Kessler. The Alpha 21264 microprocessor. *IEEE Micro*, 19(2):24–36, 1999.

[KMAC03] C. Keltcher, K. McGrath, A. Ahmed, and P. Conway. The AMD Opteron processor for multiprocessor servers. *IEEE Micro*, 23(2):66–76, March/April 2003.

[KOH+94] 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. Henessey. The Stanford Flash multiprocessor. In *The 21st International Symposium on Computer Architecture*, Chicago, IL, April 1994.

[Kre04a] K. Krewell. Intel's PC roadmap sees double. *Microprocessor Report*, 18(5):41–43, May 2004.

[Kre04b] K. Krewell. Sun's Niagara pours on the cores. *Microprocessor Report*, 18(9):11–13, September 2004.

[KST04] R. Kalla, B. Sinharoy, and J. Tendler. IBM Power5 chip: A dual-core multithreaded processor. *IEEE Micro*, 24(2):40–47, March/April 2004.

[Law] K. Lawton. Bochs. http://bochs.sourceforge.net.

[Lee87] R. Lee. *The Effectiveness of Caches and Data Prefetch buffers in Large-Scale Shared Memory Multiprocessors.* PhD thesis, University of Illinois at Urbana-Champaign, Urbana-Champaign, IL, May 1987.

[LGH94] J. Laudon, A. Gupta, and M. Horowitz. Interleaving: A multithreading technique targeting multiprocessors and workstations. In *The 6th International Conference on Architectural Support of Programming Languages and Operating Systems*, San Jose, CA, October 1994.

[LL97] J. Laudon and D. Lenoski. The SGI Origin: A ccNUMA highly scalable server. In *The 24th International Symposium on Computer Architecture*, Denver, CO, May 1997.

[LLG+92] D. Lenoski, J. Laudon, K. Gharachorloo, W. Weber, A. Gupta, J. Hennessy, M. Horowitz, and M. Lam. The Stanford Dash multiprocessor. *IEEE Transaction on Computer*, 25(3):63–79, March 1992.

[LPI88] S. Laha, J. A. Patel, and R. K. Iyer. Accurate low-cost methods for performance evaluation of cache memory systems. *IEEE Transactionctions on Computers*, February 1988.

[MCE+02] P. Magnusson, M. Christensson, J. Eskilson, D. Forsgren, G. Hallberg, J. Hogberg, F. Larsson, A. Moestedt, and B. Werner. Simics: A full system simulation platform. *IEEE Computer*, 35(2):50–58, February 2002.

[MG91] T. Mowry and A. Gupta. Tolerating latency through software-controlled prefetching in shared-memory multiprocessors. *Parallel and Distributed Computing, Special issue on shared-memory multiprocessors*, 12(2):87–106, June 1991.

[MH94] S. Mukherjee and M. Hill. An evaluation of directory protocols for medium-scale shared-memory multiprocessors. In *The 8th International Conference on Supercomputing*, July 1994.

[ONH+96] K. Olukotun, B. Nayfeh, L. Hammond, K. Wilson, and K. Chang. The case for a single-chip multiprocessor. In *The 7th International Conference on Architectural Support of Programming Languages and Operating Systems*, Cambridge, MA, October 1996.

[OR99] H. Oi and N. Ranganathan. Utilization of cache area in on-chip multiprocessor. In *HPC*, 1999.

[PP86] M. Paramarcos and J. Patel. A low-overhead coherence solution for multiprocessors with private cache memories. In *The 11th International Symposium on Computer Architecture*, Ann Arbor, MI, June 1986.

[Raz05] Raza Microelectronics, Inc. XLR processor product overview, May 2005.

[SBD+97] P. Stenström, M. Brorsson, F. Dahlgren, H. Grahn, and M. Dubois. Boosting the performance of shared memory multiprocessors. *IEEE Transaction on Computer*, 30(7):63–70, July 1997.

[SBG+02] V. Srinivasan, D. Brooks, M. Gschwind, P. Bose, V. Zyuban, P. Stenski, and P. Emma. Optimizing pipelines for power and performance. In *The 35th International Symposium on Microarchitecture*, Istanbul, Turkey, November 2002.

[SC02] E. Sprangle and D. Carmean. Increasing processor performance by implementing deeper pipelines. In *The 29th International Symposium of Computer Architecture*, Anchorage, AK, May 2002.

[SJG] P. Stenström, T. Joe, and A. Gupta. Comparative performance evaluation of cache-coherent numa and coma architectures.

[Smi82] A. Smith. Cache memories. *ACM Computing Surveys*, 14(3):473–530, September 1982.

[SPHC02] Timothy Sherwood, Erez Perelman, Greg Hamerly, and Brad Calder. Automatically characterizing large scale program behavior. In *The 10th International Conference on Architectureal Support for Programming Languages and Operating Systems*, pages 45–57, October 2002.

[SS86] P. Sweazey and A. Smith. A class of compatible cache consistency protocols and their support by the IEEE futurebus. In *The 13th International Symposium on Computer Architecture*, Tokyo, Japan, June 1986.

[SSZR05] E. Speight, H. Shafi, L. Zhang, and R. Rajamony. Adaptive mechanisms and policies for managing cache heirarchies in chip multiprocessors. In *The 32nd International Symposium of Computer Architecture*, Madison, WI, June 2005.

[Ste90] P. Stenström. A survey of cache coherence schemes for multiprocessors. *IEEE Transaction on Computer*, 23(6):12–24, 1990.

[SWCL95] A. Saulsbury, T. Wilkinson, J. Carter, and A. Landin. An argument for simple COMA. In *The 1st International Symposium on High Performance Computer Architecture*, Raleigh, NC, January 1995.

[TDJ+02] J. Tendler, J. Dodson, J. Fields Jr., H. Le, and B. Sinharoy. POWER4 system microarchitecture. *IBM Journal of Research and Development*, 46(1), 2002.

[UC 01] UC Berkeley Device Group. Predictive technology model. Technical report, UC Berkeley, 2001.

[WWFH03] R. Wunderlich, T. Wenisch, B. Falsafi, and J. Hoe. SMARTS: Accelerating microarchitecture simulation via rigorous statistical sampling. In *The 30th International Symposium of Computer Architecture*, San Diego, CA, June 2003.

[ZA05a] M. Zhang and K. Asanović. Victim Migration: Dynamically adapting between private and shared CMP caches. Technical Report MIT-CSAIL-TR-2005-064, Computer Science and Artificial Intelligence Laboratory, Massachusetts Institute of Technology, October 2005.

[ZA05b] M. Zhang and K. Asanović. Victim Replication: Maximizing capacity while hiding wire delay in tiled chip multiprocessors. In *The 32nd International Symposium on Computer Architecture*, Madison, WI, June 2005.

[ZT97] Z. Zhang and J. Torrellas. Reducing remote conflict misses: NUMA with remote cache versus COMA. In *The 3rd International Symposium on High Performance Computer Architecture*, San Antonio, TX, January 1997.