SYNERGISTIC CACHING IN SINGLE-CHIP MULTIPROCESSORS

A DISSERTATION
SUBMITTED TO THE DEPARTMENT OF ELECTRICAL ENGINEERING
AND THE COMMITTEE ON GRADUATE STUDIES
OF STANFORD UNIVERSITY
IN PARTIAL FULFILLMENT OF THE REQUIREMENTS
FOR THE DEGREE OF
DOCTOR OF PHILOSOPHY

Sarah Leilani Harris
March 2005

© Copyright by Sarah Leilani Harris 2005
All Rights Reserved

I certify that I have read this dissertation and that, in my opinion, it is fully adequate in scope and quality as a dissertation for the degree of Doctor of Philosophy.

William James Dally
(Principal Adviser)

I certify that I have read this dissertation and that, in my opinion, it is fully adequate in scope and quality as a dissertation for the degree of Doctor of Philosophy.

Mark A. Horowitz

I certify that I have read this dissertation and that, in my opinion, it is fully adequate in scope and quality as a dissertation for the degree of Doctor of Philosophy.

Christos Kozyrakis

Approved for the University Committee on Graduate Studies.

Abstract

This dissertation addresses one of the fundamental questions driving all computer architecture and design: how do we make computers faster? In this work, we explore the direction of caching systems as they are driven by developing technology, particularly the confluence of many processors onto a single die. We call these systems Chip Multiprocessors (CMPs). We explore how we can exploit the advantages uncovered by these developments – most significantly, high on-chip bandwidth and low latency – to develop a more efficient, adaptable and elegant caching system.

With increasing numbers of transistors on a chip and the subsequent convergence of many processing nodes onto a single chip, we see three dominant effects. First, because the entire system is on chip, the cost of communication between nodes is changing. In the on-chip environment, wires are cheap and area is expensive. Instead of sacrificing large amounts of area to decrease bandwidth requirements – as is often done in the multi-chip multiprocessor environment – it is better to minimize area and make use of the high on-chip bandwidth.

Second, in CMP systems we see a larger range of memory access times. The delay to memory can no longer be lumped into a single number. The physical location of the memory greatly affects the delay to retrieve the data. In previous architectures, the time to get from the processor to data was an all-or-nothing gamble. Either the data was on chip, in the processor's local cache, or it would take a long time to get the data from off-chip memory.

Finally, processors working on the same problem encourage synergy in the use and sharing of data. With a group of processors working on the same problem, data needed by one processor may have already been pulled in from main memory by a

neighboring processor. Instead of requesting data from main memory, many data requests can be serviced by the caches of neighboring nodes if that communication path is enabled.

In the following chapters we answer the following three questions: How do we organize memory—namely, cached data—to allow processors to access data at a minimal latency? What is the control of the system for allocating, locating, and communicating data most efficiently among the caches of the on-chip processors? And lastly, what are the behaviors of various applications and are they conducive to synergistic caching?

We introduce the synergistic caching (SC) system to answer these questions. We analyze a set of applications and show that sharing can be used to alleviate processor stall cycles due to memory stalling. We show that the SC system is most beneficial when there are large amounts of sharing in an application, when the application is capacity-limited, and when the application shows low latency tolerance.

We examine a set of five benchmarks and show that 15-70% of processor cycles are spent stalling on data and that 15-99% of accesses are to shared data. Across these same applications, we show that, using the SC system, 7-72% of accesses that would have been supplied by main memory can instead be serviced by a neighboring cache. The SC system subsequently reduces average memory access time by up to 43% and improves performance by up to 28% for the same applications as compared to an L1 caching system.

We examine various duplication modes: *beg*, *borrow*, and *steal*. Across these duplication modes, we show that *overall hit rates* can vary as much as 10%, average memory access time varies by as much as 75%, and execution time varies by as much as 50%. We also compare the SC system to existing shared-L2 caching systems. As compared to a shared-L2 caching system using comparable area, the synergistic caching system exhibits up to a 21% higher hit rate, has up to a three times improvement in average memory access time, and has up to a 92% speedup in execution.

In the following thesis, we show that synergistic caching takes advantage of the on-chip environment and minimizes the chip area of the caching system by utilizing data sharing among processors to improve overall application performance.

Acknowledgements

Contents

Abstract **iv**

Acknowledgements **vi**

1 Introduction **1**

1.1 The Challenges and Opportunities 1

1.2 Synergistic Caching 3

1.3 Contributions 5

1.4 Roadmap 6

2 Background **8**

2.1 Chip Multiprocessors 8

2.2 Related Work 9

2.2.1 Directory-Only 10

2.2.2 Bus-Based 10

2.2.3 Hybrid 11

2.2.4 Other Coherence Mechanisms 11

2.2.5 Data Placement 12

3 The Synergistic Caching System **16**

3.1 Functionality 16

3.2 Architecture 17

3.3 Baseline Hardware 20

3.3.1 Additional Hardware 23

3.3.2 Contention 24
3.3.3 Buffer Contention 26
3.3.4 Protocols 26
3.3.5 Coherency and Consistency 32
3.3.6 Summary 33
3.4 Synergistic Caching Variations 34
3.4.1 Duplication Policies: Beg, Borrow, Steal 34
3.4.2 Capacity 35
3.4.3 Performance Trade-offs 37
3.4.4 Duplication Hardware 38
3.5 Area 39
3.6 Benefits of Synergistic Caching 40
3.6.1 Sharing 40
3.6.2 Capacity-limits 43
3.6.3 Latency Tolerance 43
3.7 Summary 47

4 Performance Results 48

4.1 Experimental Set-Up 48
4.2 Applications 50
4.2.1 Memory Stalling 51
4.2.2 Shared Data 52
4.2.3 Retrieving Shared Data 56
4.3 Performance Results 57
4.3.1 Miss Rates 57
4.3.2 Average Memory Access Time 62
4.3.3 Execution Time 65
4.3.4 Summary 69
4.4 Duplication Modes 69
4.4.1 Hit Rates 70
4.4.2 Average Memory Access Time 71

4.4.3 Execution Time 76
4.5 Summary 80

5 Shared-L2 Comparison 81

5.1 Area 82
5.2 Performance 83
5.2.1 Hit Rate 84
5.2.2 Average Memory Access Time 86
5.2.3 Execution Speedup 89
5.3 Summary 92

6 Conclusions and Future Work 93

A Glossary 97

Bibliography 101

List of Tables

3.1 Buffer Functionality. 21
3.2 Memory Request Priorities. 32
3.3 Cluster Cache Capacities. 35
3.4 Cache Sizes. 40
3.5 Sharing Categories. 43

4.1 Benchmarks. 50

List of Figures

1.1 CPU vs DRAM speeds since 1980. 2
1.2 Synergistic Caching Functionality. 4
1.3 Synergistic Caching System – Two-Node Cluster. 5

3.1 A Single Node of the Synergistic Caching System. 18
3.2 A CMP Synergistic Caching System with 64 nodes. 18
3.3 A 2-Node Cluster. 19
3.4 Synergistic Caching Buffer Details. 20
3.5 Synergistic Caching Additional Hardware. 24
3.6 Write Request Protocol. 29
3.7 Read Request Protocol. 30
3.8 Duplication Modes. 35
3.9 Multi-threading. 42
3.10 Load-use distance. 44
3.11 Multi-threading. 45

4.1 Synergistic Caching Experimental Set-up. 49
4.2 Percentage of Stall Cycles. 52
4.3 Shared Data. 53
4.4 Shared Data Details. 54
4.5 Shared Reads/Writes. 55
4.6 Miss Rates for the L1 and Synergistic Caching Systems. 58
4.7 Hit Rate Details. 59
4.8 Percent Decrease in Main Memory Accesses. 60

4.9 Average Memory Access Time. 62
4.10 Cluster Cache Average Access Time (CC-AT). 64
4.11 Cluster Cache Average Access Time (CC-AT) Details. 65
4.12 Synergistic Caching and Traditional Caching Execution Times. 66
4.13 Execution Speedup of the Synergistic Caching System over the Traditional Caching System. 66
4.14 Synergistic Caching System Detailed Execution Time. 68
4.15 Hit Rates of the Duplication Modes. 71
4.16 Average MAT across duplication modes. 72
4.17 Normalized average MAT for each duplication mode. 73
4.18 Cluster-cache Average Access Time (CC-AT) Across Duplication Modes. 74
4.19 Port Contention Cycles at the Cluster Cache. 74
4.20 Instruction Mix. 75
4.21 Execution Speedup for Duplication Modes as Compared to an Independent-L1 Caching System. 76

5.1 A 2-node Cluster of an L2 Caching System. 82
5.2 Area estimates. 83
5.3 Hit Rates for the Synergistic and L2 Caching Systems. 84
5.4 Hit rate for the L1 caching system. 85
5.5 MAT Comparison of SC and L2 Systems. 86
5.6 Average access times for the cluster and L2 cache. 88
5.7 Execution time of synergistic and shared-L2 caching systems. 90
5.8 Synergistic Caching Execution Speedup as Compared to a Shared-L2 System. 91

Chapter 1

Introduction

Over the past several decades, advances in technology—including increases in the number of transistors per chip—have allowed for dramatic advances in computer architecture. In this thesis we look specifically at the trend of placing multiple processing cores on a single chip. Parallel processors are now becoming a viable option for mainstream computer systems[29][25][3].

In this chapter we discuss the challenges facing current computer architects and the opportunities afforded by single-chip multiprocessor (CMP) systems. We introduce synergistic caching as a means of addressing a challenge of these systems by effectively utilizing the characteristics of CMP systems. We then discuss the overall contributions of this work and provide a roadmap for the remainder of the thesis.

1.1 The Challenges and Opportunities

With the emergence of chip multiprocessor systems, the cost of communication is changing. In the on-chip environment, wires are cheap and area is expensive. This is in contrast to the multi-chip systems of the past where bandwidth between chips was the bottleneck and on-chip area was relatively abundant. With these developments, while

many of the challenges of building fast, reliable computer systems remain the same, the trade-off equation changes. Two of the main challenges facing computer architects are (1) designing a memory system that can keep up with increasing processor speeds, and (2) balancing memory bandwidth demand with the chip area used to store local data.

Figure 1.1[1] shows processor speeds versus DRAM speeds over time from around 1980. Processor speeds are increasing, as shown by the top line, according to Moore's law at about 60% per year while memory speeds are increasing at a much slower rate of about 10% per year as shown by the bottom line [6]. So, while the current gap between processor speeds and memory speeds is large, the worst news is that this gap is increasing [34].

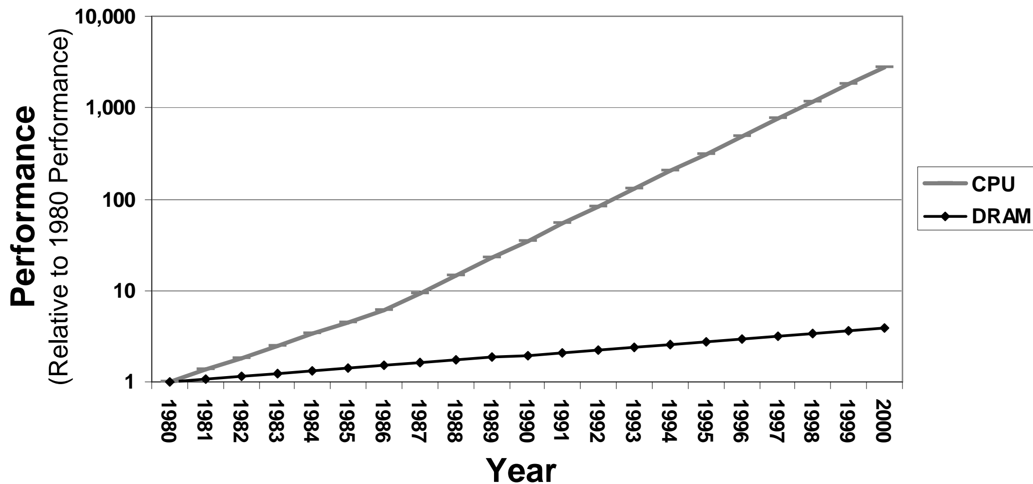

Figure 1.1: CPU vs DRAM speeds since 1980.

Caching is a strategy that has been introduced to bridge the gap between processor and memory speeds. With the introduction of caches, the processor can access most data at the processor speed and not at the slower DRAM speed.

[1] This figure is adapted from [6].

Synergistic caching extends the traditional caching strategy to the chip multiprocessor environment. It uses both the characteristics of the CMP environment and those of the multi-threaded applications running on these systems to extend traditional caching strategies.

The characteristics of CMP systems that synergistic caching leverages are high on-chip bandwidth and low latency to neighboring nodes. Delay to neighboring nodes is small and bandwidth between nodes is limited by wire pitch, and not the number of I/O pins, as in multi-chip systems. In addition, multi-threaded applications running on CMPs offer large amounts of data sharing between threads. Thus, a thread requesting data on one node may be able to retrieve that data from a neighboring node instead of from main memory. We talk in detail about the sharing characteristics of multi-threaded applications in Chapter 3.

In the next section, we give an overview of synergistic caching strategies that use the characteristics of the CMP environment and multi-threaded applications to decrease the processor-memory gap and to use on-chip area efficiently.

1.2 Synergistic Caching

We introduce *synergistic caching* to address the architecture challenge of designing a fast memory system while balancing memory bandwidth demand with chip area. Synergistic caching addresses these challenges by increasing effective memory performance and minimizing the area devoted to the caching system.

Caching was introduced to decrease average memory access time by exploiting multiple use of data by a single processor. Synergistic caching extends traditional caching to the multiprocessor environment by enabling multiple use of data by *multiple* processors. It allows a processing node to retrieve data from a neighboring cache (through the on-chip network) instead of retrieving the data from main memory.

When multiple processors are added to the single-chip system, a processor has three locations from which to retrieve data: its local cache, main memory, or the neighboring cache. Figure 1.2 depicts the functionality of a 2-node synergistic caching system.

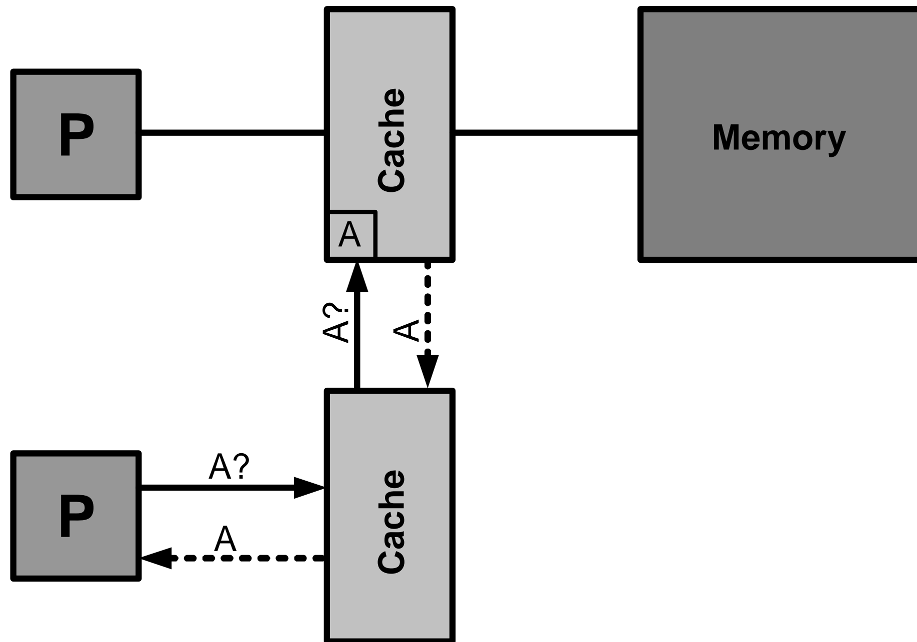

Figure 1.2: Synergistic Caching Functionality.

The solid lines are requests for data, and the dotted lines are replies. In the figure, data A is requested by the lower processor. The lower processor requests data A from the above processor's cache before requesting it from memory. Data A is held in the neighboring cache, and the processor supplies the data to the requesting processor, as depicted by the dotted lines.

Upon an L1 miss in a synergistic caching system, the requesting processor forwards the request to the neighboring cache instead of to main memory. Only upon missing in the neighboring cache is the request serviced from main memory.

Figure 1.3 shows a two-node cluster of the synergistic caching system. Each

processing node has its own cache controller and L1 cache. Each node is also connected to the network via a network controller. If a request misses in the local cache, the request is sent across the network to the neighboring node. There are variations on timing and duplication modes that we discuss in Chapter 3.

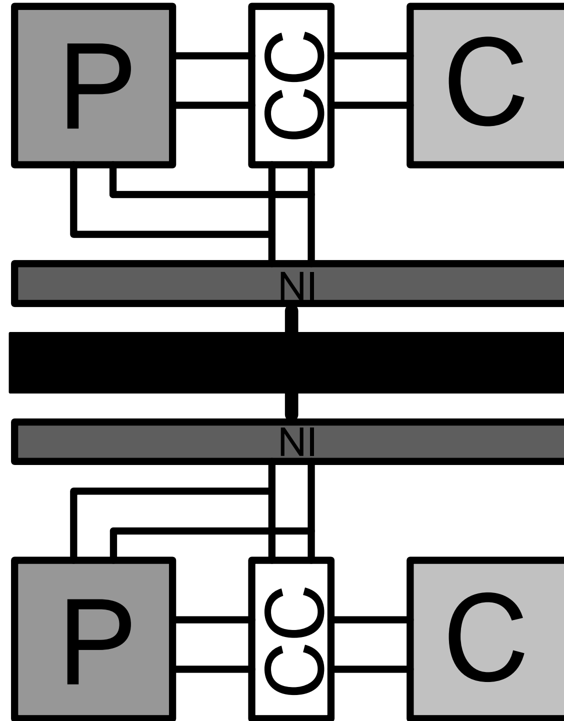

Figure 1.3: Synergistic Caching System – Two-Node Cluster.

1.3 Contributions

In the remainder of this thesis, we define the synergistic caching system. We define the architecture, the hardware and the protocols that enable communication between caches of neighboring nodes. We also show several duplication variations of the baseline synergistic caching system. These modes are the *beg*, *borrow*, and *steal* modes.

We examine multi-threaded application characteristics, namely: sharing, latency

tolerance, and data locality. We define how these application characteristics affect performance of a synergistic caching system. We show that synergistic caching is advantageous when an application is limited by cache capacity, exhibits large amounts of sharing, and has low latency tolerance. Lastly, we compare the synergistic system with existing shared level-2 caching systems. In summary, the contributions of this thesis are that we:

- Developed the synergistic caching architecture
- Developed synergistic caching protocols.
- Defined duplication variations: *beg*, *borrow*, and *steal*.
- Defined application characteristics that determine application performance, namely: percentage of memory stall cycles, amount of shared data, latency tolerance, and data locality.
- Compared synergistic caching with independent-L1 caching and shared-L2 caching.

1.4 Roadmap

In the next chapter we begin by describing the advantages of CMPs and by discussing other work related to synergistic caching. In Chapter 3, we introduce synergistic caching in detail. We introduce the architecture and define the protocols used by the system. We also discuss duplication policy variations on the basic synergistic caching system.

In Chapter 4, we describe the simulation environment we designed to test the system and the set of benchmarks we ran on the system. We compare the synergistic caching system to a system with independent L1 caches. We follow this with a

comparison to a shared-L2 caching system in Chapter 5. Chapter 6 gives conclusions and describes future directions for further study related to synergistic caching systems.

Chapter 2

Background

Moore's law predicts exponential increases in the number of transistors per chip with transistor density doubling every 18 months [18]. Because of this trend, systems that several decades ago were distributed across several chips or even boards now fit on a single die[1].With further increases in the number of transistors, in the last 10 to 15 years, we have seen the emergence of multiple processing cores on a single chip [3][29][25][8]. These chip multiprocessor systems (CMPs) offer new challenges — as well as opportunities.

2.1 Chip Multiprocessors

While multi-chip multiprocessors supply more computational power than a single processing node, they suffer from the drawbacks of low bandwidth and high latency between nodes. These factors discourage communication between nodes in multi-chip systems. With the emergence of chip multiprocessors, the equation balancing the

[1]For example, the VAX 11/780 released in 1977 required several circuit boards for its CPU, performed at one million instructions per second, and occupied the better part of a room [14]. On the other hand, the Pentium processor, first released in 1993, fits on a single chip, costs three orders of magnitude less than the VAX 11/780, and performs at about 100 MIPS.

trade-off between bandwidth demand and chip area has changed. CMP systems offer high bandwidth and low latency between neighboring nodes that is limited by wire pitch, not the number of I/O pins as it is in multi-chip multiprocessors. With CMP systems, the relative abundance of bandwidth and lower latency between nodes makes local communication relatively cheap. On-chip area then, instead of bandwidth, is at a premium, which suggests one look at using the high on-chip bandwidth to reduce the area requirements.

In traditional systems caches are used to reduce bandwidth to main memory by placing small memories close to the processor that hold frequently used data [27]. Synergistic caching extends traditional caching strategies to the multiprocessor environment. Synergistic caching takes advantage of sharing in multi-threaded applications and the CMP environment (high on-chip bandwidth and low latency) to decrease memory access time and the amount of on-chip area used for the caching system.

2.2 Related Work

There has been large interest in parallel processors over the last half century[2]. The main issue in building multiprocessor systems is ensuring that the processors work well together. In order to do this, each processor in a multiprocessor system needs to have memory that is local to the node to store frequently accessed data. This has been traditionally implemented using a cache and, often, a portion of the main memory. However, in order to ensure that each processor sees the same picture of memory, the distributed shared memory and caches must be kept coherent. Thus, most of the work on multiprocessor cache organization focused on a specific coherency mechanism. These fall under three broad categories:

- Directory only schemes such as COMA ([31],[23],[20]) and some CC-NUMA machines ([11],[12]).

- Bus snooping mechanisms ([22],[21], [5]).
- Hybrid solutions where the directory deals with clusters of bus-snooping processing elements ([13],[3]).

In the following sections we discuss how each strategy addresses the coherency issue and how synergistic caching proposes to use the underlying mechanisms already in place for coherency to aid in the sharing of data. We also discuss the issues of data placement and bandwidth usage as related to synergistic caching methods.

2.2.1 Directory-Only

Directory-only machines hold location and sharing information at a central location– on the home node. This class of machines generally does not allow sharing of cache resources between processors or take network locality into account.

CC-NUMA does allow for 3-hop transactions, in which a processor's memory request is serviced by another's cache, when the remote processor is the owner of the relevant cache line [12]. But this mechanism serves as an optimization for guaranteeing correctness without continuous writebacks to main memory. When implemented upon a hierarchical network COMA does provide some measure of locality if the COMA directory tree corresponds to the network hierarchy.

2.2.2 Bus-Based

Bus-based multiprocessors connect all or a subset of processors onto a single bus. This class of machines has an advantage in that each processor is aware of the transactions of all other processors.

Several studies explored the design space of sharing cache resources on such a system. Work done at Kendall Square Research suggested several processors share a

banked L1 cache interleaved on cache lines [22]. This enables a large first level cache at the cost of higher latency access for all loads.

Nayfeh et al. revisited the same design and also suggested a configuration in which each processing element contained a non-shared L1 cache while the L2 cache was interleaved among all processors in the cluster [21]. This design was then chosen to implement the Hydra chip multiprocessor [5].

2.2.3 Hybrid

The hybrid hierarchical snooping directory of the DASH architecture connects 4-processor clusters [13]. Each cluster contains four processing elements along with their L1 caches, a shared L2 cache, and a directory controller which had a *Remote Access Cache*. Each miss in a local L1 results in a query to the shared L2 cache, and if that results in a miss as well the request is placed on the cluster bus to be serviced directly by one of the other four caches (three L1s and the RAC).

Piranha follows a similar methodology, except that the L2 cache includes duplicates of the L1 tags. This allows for complete exclusion and increases the effective capacity of the on-chip caches [3].

2.2.4 Other Coherence Mechanisms

Martin et al. introduced a new method for cache coherence in shared memory multiprocessors called token coherence [17]. This scheme eliminates the need for a shared bus or virtually shared bus, and they argue that it is faster and simpler than snooping or directory protocols. In the token coherence protocol, a processor must collect at least one token to read a data line, and it must collect all tokens to write the data. It uses token counting and persistent requests to enforce correctness [17]. While cache coherence is a related concern, in this thesis we treat coherency as an orthogonal issue

to cache organization and explore configurations that are not available in the above schemes.

2.2.5 Data Placement

In research most closely related to our work, several other groups have explored data placement in caching systems. The most notable of these are Pirahna[3] the NUCA system [9] and the shared processor-based split L2 design [15].

Piranha follows a similar methodology to the hybrid directory approaches except that the L2 cache includes duplicates of the L1 tags[3]. As mentioned above, this allows for complete *exclusion*[2], thus increasing the effective size of the on-chip caches. The Piranha system takes advantage of sharing among eight processing cores by sharing data through the shared L2 cache.

As with Piranha, Liu et al. explored the use of L2 caches for enabling sharing [15]. They evaluated the difference between private L2 and address-interleaved shared L2 designs in a CMP system and proposed a new design called the shared processor-based split L2 [15]. To alleviate redundancy of a private L2 cache and to enable sharing, they assign banks of a shared-L2 cache to specific processors. They propose that this new design avoids the drawbacks of the private and shared L2 designs, namely imbalances in non-uniform demand and interference, respectively.

They aim to make their system run well both when there are large amounts of shared data as well as when there is little sharing between nodes. They use a bus-based system where all of the L2 banks reside on the other side of the bus. They use static partitioning such that banks are assigned to one or more processors. This mapping can be changed between runs or dynamically during run-time by downloading the configuration information using an operating system call. After a miss in the L2

[2]Exclusion means that data held in the L1 cache is not also held in the L2 cache. Thus, the data held in the L1 cache or L2 cache are exclusively held in the respective level of the cache hierarchy.

bank(s) assigned to a processor, the processor then checks the other (non-assigned) L2 banks. However, since they aim to minimize off-chip memory accesses, they send the main memory request only after a miss in all of the L2 banks. Synergistic caching offers the options of sending the request either after or concurrently with the cluster request. The data is kept in the bank of the processor that most recently accessed it.

The last data placement strategy, non-uniform cache access (NUCA), was defined by Kim et al. [9]. They defined dynamic allocation methods for placing commonly-accessed data in the closest block of an L2-cache. They tested their system on a subset of SPEC2000 super-scalar applications. They showed performance improvements by exposing the range of delays offered by the different banks of a large L2 cache. They found that performance increased the most when the closest banks of the L2 cache held the most frequently accessed data. Their proposed design gives lower access times than conventional L2 caches, promises to be scalable as cache sizes increase, and enables performance stability for varying sizes of working sets. NUCA focuses on taking advantage of the physical locality of data on super-scalar applications running on a single processor. Their system, however, does not explore the effects of sharing between processors.

Related to coherency and data placement is the use of overall bandwidth. Increasing coherency traffic can slow down the overall system and placing data close to requesting nodes ideally decreases the network traffic. The developers of Niagara, a Chip Multi-Threaded (CMT) processor, propose to use Simultaneous Multithreading (SMT) along with thread speculation to increase processor performance [10, 28]. Each of the eight SMT processors in their design have a private level-1 cache and share a single level-2 cache through a crossbar switch. This simplifies the coherence protocols by filtering them through the L2 cache. SMT techniques can hide some of the latency of outstanding memory requests, however the demand on the level-1 cache also increases. This system proposes to use the bandwidth/area trade-off for

speculation as well as executing multiple threads on a single core[28].

Synergistic caching proposes to employ the mechanisms that are already in place for coherency—the existing directories and interconnection between processor nodes—to enable sharing between the caches of neighboring nodes. However, as opposed to the Niagara system, SC proposes to use the additional bandwidth afforded by the on-chip system for accessing neighboring nodes instead of executing speculative threads. The hybrid hierarchical snooping directory, where processors can access data from neighboring nodes, resembles synergistic caching most closely. However, instead of being connected by a shared bus, in the SC system, all nodes are connected via the on-chip network.

The Piranha system is similar to an SC system since a group of processors share data, but SC eliminates the need for the shared-L2 cache and allows sharing directly at the first-level cache. The trade-off is that the requests to main memory are no longer filtered through the L2 cache. Piranha also hard-wires the number of sharing processors while SC allows the number to be flexible. As with SC, Piranha also uses an on-chip network, the *intra-chip switch (ICS)*, instead of a bus for inter-node sharing through the on-chip L2 cache. However, the switch is dedicated to sending L2 requests whereas the synergistic caching system uses a single on-chip network for both local requests and across-chip requests[3].

The lookup strategy used by Liu et al. in the L2 cache is similar to SC's cluster lookup using L1 caches. In their system, after a miss in the L2 bank(s) assigned to a processor, the processor then checks the other (non-assigned) L2 banks. However, since they aim to minimize off-chip memory accesses, they send the main memory request only after a miss in all of the L2 banks. Synergistic caching offers the options of sending the request either after or concurrently with the cluster request.

The duplication policy they used by Liu et al. is similar to our *steal* mode (see

[3] In practice, an SC system could also be implemented using a separate, dedicated network.

Chapter 3). However, they do not explore additional duplication policies as we do for synergistic caching in the same chapter.

Both SC and NUCA focus on taking advantage of the physical locality of data, but the key difference between the two approaches is that NUCA focuses on super-scalar applications running on a single processor. Their system does not address the effects of sharing between processors as we will explore using synergistic caching methods.

The arena of data placement and sharing has been explored by many groups over about the last several decades. Synergistic caching proposes to explore the space of exchanging bandwidth for storage by reducing the area needed for the overall caching system by eliminating the need for the L2 cache. Requests that would have been serviced by the L2 cache are instead serviced by neighboring caches in the cluster. The hardware needed for enabling this sharing is largely in place due to the need for coherency in chip multiprocessors. We also propose using many simple processing cores instead of a few complex cores—i.e. multiple-issue, out-of-order, or speculative. We examine this space of the area/bandwidth trade-off and the use of simple processors in the rest of this thesis.

While data placement, clustered cache organizations, and sharing have been explored in the past, current trends point to tighter integration and even higher densities of nodes. These trends point to the use of on-chip interconnection networks instead of buses and increased communication between nodes to take advantage of locality. This, in turn, allows more flexibility and enables novel configurations. In the following chapters we show how synergistic caching utilizes the CMP environment to decrease memory access time, efficiently use chip area, and improve performance.

Chapter 3

The Synergistic Caching System

In this chapter we introduce the synergistic caching system in detail. We show how synergistic caching enables communication between neighboring processors by trading bandwidth for chip area. While traditional caching has been a means of alleviating the long delay of accessing main memory by keeping frequently accessed data temporally and physically close to the processor, we show how synergistic caching extends traditional caching to the on-chip multiprocessor environment by allowing data shared by many processors to be quickly and frequently accessed by all neighboring processors. We describe the architecture of the synergistic caching system by beginning with a functional description. We then describe the baseline architecture and introduce variations to that baseline system.

3.1 Functionality

In a traditional caching system, a processor accesses data from two sources: the local cache[1] or main memory. The local cache, typically made of Static Random Access

[1]There may also be additional levels of caching. We discuss these configurations in Chapter 5 and compare them to the synergistic system.

Memory (SRAM), holds data that has been previously accessed by the processor. The cache holds both the address of previously accessed data, the tags, and the data at those memory locations. The size of the first-level cache, the L1 cache, is kept small to make the access time on the order of one processor clock cycle. Main memory, on the other hand, has an access time on the order of 10's to 100's of clock cycles.

In a multiprocessor system, a processor now has potentially three locations from which to retrieve data: its local cache, main memory, or the neighboring cache. Upon an L1 miss in a synergistic caching system, the requesting processor forwards the request to the neighboring cache instead of to main memory. Only upon missing in the neighboring cache does the request get serviced by main memory.

In the following sections we examine the system architecture that enables a node in the synergistic caching system to access data in a neighboring node's cache and introduce the architecture and mechanisms for implementing synergistic caching.

3.2 Architecture

The synergistic caching architecture builds upon the basic architecture of a traditional caching system and then extends its capabilities to allow sharing between caches of neighboring nodes. We define the protocols, the consistency and coherency models used, and the additional hardware required for synergistic caching.

Figure 3.1 shows a single node of the synergistic caching system. Similar to a traditional caching system, each processor has a cache controller and its local cache, the level-1 (L1) cache. The processor and cache controller are connected to the network interface.

In a CMP system, many of these nodes are placed on a single chip and are connected by an on-chip network. Figure 3.2 shows a 64-node CMP system, an 8x8 node configuration. Each node consists of a processor (shown in dark gray), a cache (shown

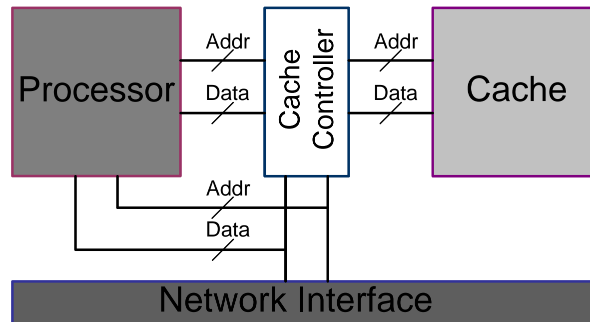

Figure 3.1: A Single Node of the Synergistic Caching System.

in light gray), and a cache controller (shown in white). All of the nodes are connected by the on-chip network, depicted in black in Figure 3.2. Each node also has its own network interface. This multi-node system is broken up into *clusters*, a number of nodes sharing data at the first-level cache. The nodes in a cluster communicate via the on-chip network making it a logical, but not physical, cluster.

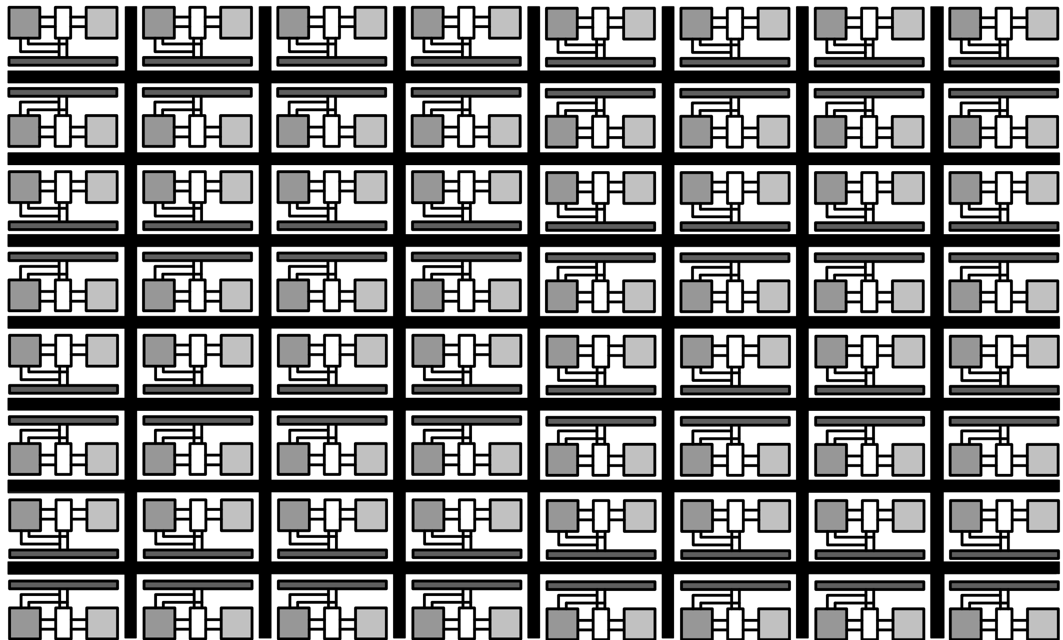

Figure 3.2: A CMP Synergistic Caching System with 64 nodes.

Figure 3.3 shows a synergistic caching cluster with a cluster size of two. Within a

cluster, each processor's cache has direct access to its own cache as well as access to a number of neighboring processor caches through the on-chip network. We call this group of caches the *cluster cache*.

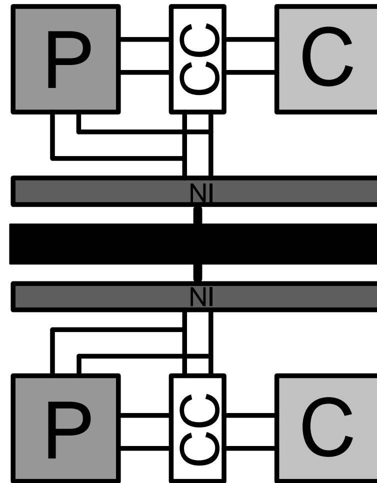

Figure 3.3: A 2-Node Cluster.

Upon a miss in a processor's local cache, the request is sent across the network to the neighboring node's cache. This is called a *cluster cache request*. Regardless of cluster size, upon a miss in a processor's L1 cache,the request is sent to all other caches in the cluster[2]

If the data are located in the cluster cache, the data are returned across the network to the requesting node. The data are used by the processor and possibly updated in the requesting node's cache[3].

This interaction is similar to pre-fetching. The shared data line held in the supplying node's cache was requested from main memory before it is needed by the requesting (neighboring) processor. However, the pre-fetch is synergistic in that the prefetching processor (the supplying node) is also benefiting from the data fetch[4][22].

[2]Variations on this algorithm are possible. For example, the request could be sent serially to each node in the cluster, or a look-up table of most recent cluster hits could be kept.

[3]We discuss duplication policies further in Section 3.4

We call this *synergistic pre-fetching*.

3.3 Baseline Hardware

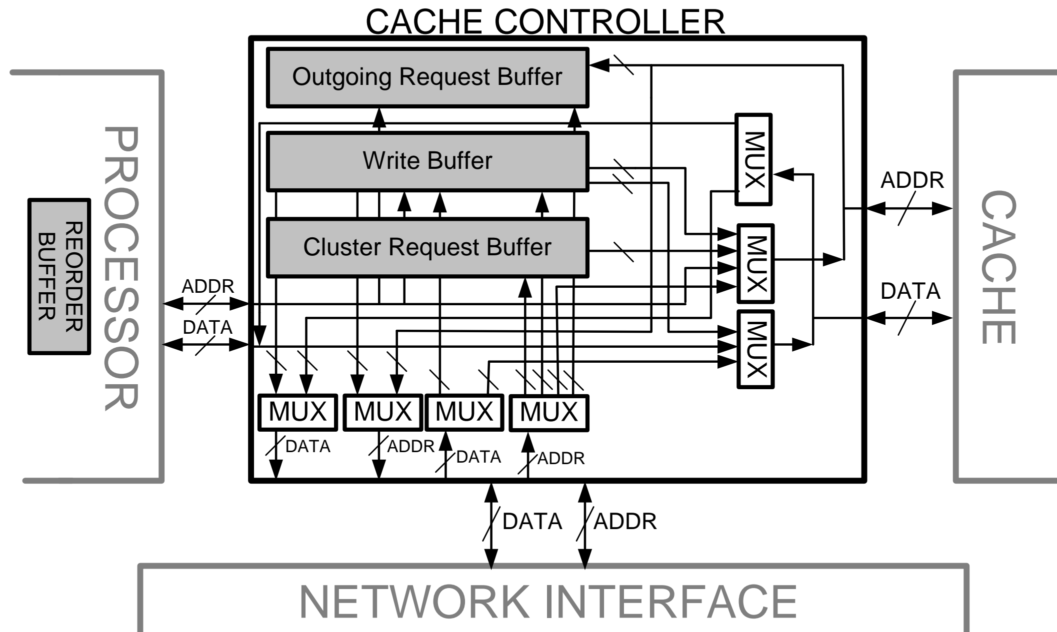

Figure 3.4: Synergistic Caching Buffer Details.

Figure 3.4 depicts the baseline hardware of a single node of the synergistic caching system. The figure shows the cache controller arbitrating between the processor, the cache, and the network interface. The system includes several buffers and multiplexers that are needed for the arbitration of resources. The buffers are: the *outgoing request buffer*, the *re-order buffer*, the *write buffer*, and the *cluster request buffer*. Table 3.1 describes the buffers in the system, their function, location, and connections with surrounding functional units.

Table 3.1: Buffer Functionality.

Buffer Name	Functionality	Connections
Outgoing Request Buffer	Requests sent across the network are stored in the buffer until the requests are satisfied. Future requests to the same cache line are not sent across the network but wait for the previously sent request to be answered.	Cache, Network
Re-order Buffer	This is the traditional re-order buffer used in current microprocessors. It allows memory requests to complete out of order and enables a non-blocking memory system. Because data delays vary for each data address, the re-order buffer keeps track of which came first and makes sure the processor sees the data in an order consistent with the request ordering.	Processor
Write Buffer	If the cache port is already being used, writes are kept in the write buffer. When the port becomes available, the writes are updated in the cache. When the processor checks the cache for data, it simultaneously checks the write buffer for the needed data	Processor, Cache, Network
Cluster Request Buffer	Upon a request from a neighboring node, if the port is not available, the cluster request is stored in this buffer. When the port becomes available, the cluster request is serviced.	Cache, Network

Only the *cluster request buffer* is an addition to the synergistic caching system. The other buffers are already needed in a traditional caching system. The *outgoing request buffer* may be an optimization to some existing systems. The data address requested from or written to the cache can come from one of four locations: the *write buffer*, the *cluster request buffer*, the processor, or the network. Similarly, there are three locations from which the data can be supplied: the *write buffer*, the processor, or the network. There will never be write data in the *cluster request buffer*. Invalidates sent from other nodes or clusters will go directly to the *write buffer* or the cache, if the cache port is available on that given cycle[4].

Data supplied by the cache will be returned to the processor or sent across the network to a requesting cluster node. Upon a miss in the level-1 cache, the read request is added to the *outgoing request buffer* and the request is sent across the network to the cluster caches.

Any incoming request checks both the *outgoing request buffer* as well as the *write buffer* for a match while simultaneously checking the cache. If there is a match in the *write buffer*, the data are returned from the *write buffer*. If there is a match in the *outgoing request buffer*, an additional request for the cache line is *not* sent onto the network. Thus, only a single request to a cache line is outstanding at any given time.

Requests sent over the network can come from either the cache or from the *write buffer*. Requests from the cache have priority over requests from the *write buffer*.

The network imposes a natural ordering on memory replies and cluster replies. Requests or replies received from the network occur for one of several reasons: (1) a cluster or non-cluster node invalidates a cache line held by this node, (2) a cluster returns data requested by this node, (3) a cluster node is requesting data from

[4]As described below in the section on coherency and consistency, we use a relaxed consistency model.

this node's cache, or (4) main memory sends data requested by this node. The request/reply from the network is received by the cache—if the cache port is not being used by the owning processor or by a previous cluster request. If the cache port is occupied, the request/reply is received by the *write buffer*—if the reply is an invalidate or reply from memory, or by the *cluster request buffer*—if the request is from a neighboring cache.

The reply also checks the *outgoing request buffer* for a match. If the corresponding request is not located in the *outgoing request buffer*, the reply is dropped. This occurs when a previous cluster reply has supplied the requested data.

3.3.1 Additional Hardware

The synergistic caching system requires little additional hardware beyond that of a traditional L1 caching system. Figure 3.5 shows the synergistic caching system hardware for a single node, with the additional hardware needed by synergistic caching highlighted.

The level-1 caching system requires an outgoing request buffer, a re-order buffer in the processor, and a write buffer. The only additional buffer required by the synergistic caching system is the cluster request buffer.

Synergistic caching also requires additional inputs to the multiplexers at the port of the cache on the address lines. These multiplexers determine, depending on priorities, whether a buffer, the processor, or the network gains control of the address lines—the cache port—for any given processor cycle. There is also an additional multiplexer required at the input to the network interface. This multiplexer determines which buffer or cache is sending data onto or retrieving data from the network.

Thus, there is minimal added hardware and delay in a synergistic caching system. The only critical path slow-down is in the addition or expansion of the multiplexers.

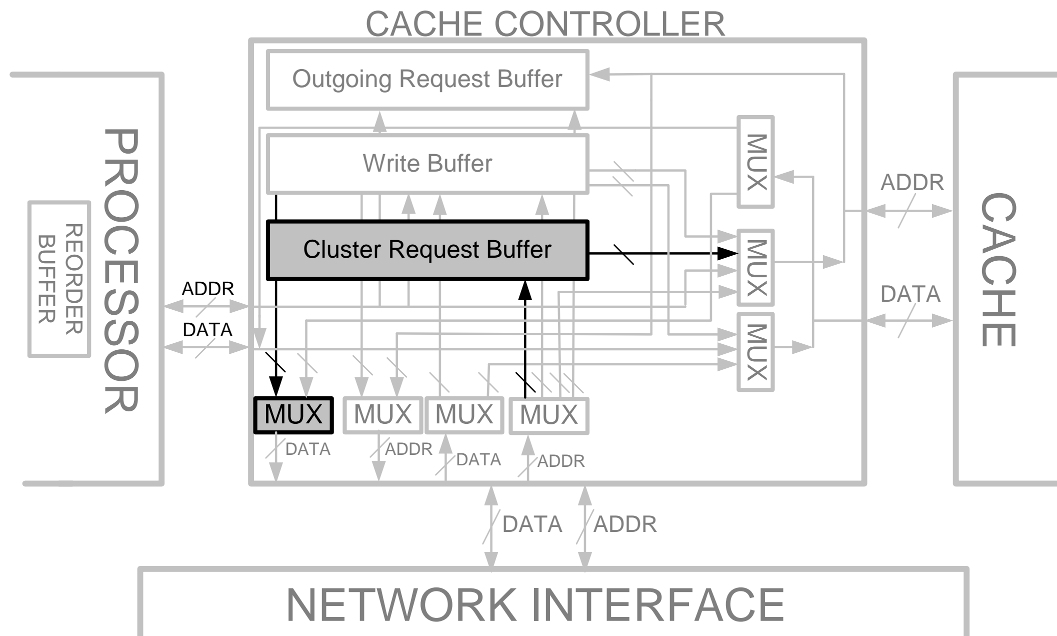

Figure 3.5: Synergistic Caching Additional Hardware.

3.3.2 Contention

As in any multiprocessor system, several sources of contention exist in the synergistic caching system. The most notable are: port contention at the level-1 cache, network contention, and buffer contention.

Port Contention

In a single-ported synergistic caching system, if the neighboring processor is already making a request to its own cache, the cluster request stalls until the port becomes available. The cluster request is serviced only *after* the primary request, the request from its own processor, is serviced.

This concept is extended to any number of ports on the cache. The cluster request is stalled when all of the ports are being used by the owning processor. Several ways exist to minimize conflicts on the cache ports, as discussed in [7][24][33][32][19]. These

include such methods as adding multiple ports, cache duplication, and using multiple banks. These methods potentially decrease the contention on a cache port, but they also incur costs such as increased area and power. As methods for alleviating this contention are not the focus of this thesis, we refer to the sources mentioned above for further inquiry. For the synergistic caching system modeled here, we account for contention at the ports of the cache by stalling the cluster request if the owning node is using the cache port. This adds potential delay in the servicing of cluster requests. However, we find that the additional latency due to port contention is minimal, as we will discuss in Chapter 4.

Network Contention

The second source of possible contention is the network. Contention in the network can cause delay due to resource contention at the virtual channels, network buffers, or network ports. Even though a cache port may be available on a neighboring node's cache, if the virtual channel between those nodes is not available, the request is blocked. However, since the cluster requests are local, as opposed to across-chip, the global contention is kept small. This keeps the imposed contention on innocent nodes, nodes not involved in the request, minimal.

Two approaches exist for cluster requests sent across the network. The first is to use the existing on-chip network to send and receive cluster requests. The second approach is to use a separate local network to send and receive requests. This private network would only connect the nodes in a single cluster.

The advantages of the first approach are simplicity and flexibility. There is no need for the additional hardware and cluster sizes are not hard-wired at the time of fabrication. Depending on the application characteristics, the cluster size can be modified at run-time or at compile-time. However, the disadvantages of sharing the existing on-chip network are that the cluster requests must compete with existing

network traffic for network resources. In turn, the cluster requests may slow down non-cluster traffic.

The second approach—where clusters use a separate local network for cluster requests—overcomes the disadvantage of competing network usage but demands additional hardware for the separate sub-network. It also hard-wires the size of the cluster at fabrication time.

In our simulations, we use the first approach to allow the number of nodes in the cluster to be adaptable and to minimize complexity of design. We expected the added latency due to contention on the global network to be small, and as we show in Chapter 4, this proved to be the case. In the results shown, network contention on the virtual channels is taken into account. Simulations where network contention on the virtual channels, buffers, and routers was taken into account showed similar results.

3.3.3 Buffer Contention

The third source of contention is at the buffers in the system. As stated above, these buffers are: the *outgoing request buffer*, the *re-order buffer*, the *write buffer*, and the *cluster request buffer*, as previously described in Table 3.1. The protocols section below describes how contention for buffer resources is resolved.

3.3.4 Protocols

Contention of resources in the synergistic caching system, especially within the cluster, also demands the implementation of protocols. Because there are multiple requesters for a single resource, for example a cache port, protocols must be implemented to arbitrate for these cluster resources. In this section we discuss the write and read request protocols and define request priorities.

Write Requests

On a write request from a processor, invalidates are sent to nodes holding the data, both nodes inside and outside the cluster. There are two options for performing this invalidate: *broadcast invalidate (BI)*, and *target invalidate (TI)*. With BI, invalidate requests are sent to all clusters. Each cluster has a local lookup table (LUT), the *cluster cache lookup table* (CC-LUT), that defines which nodes, if any, hold the data. The invalidate is then forwarded to those nodes holding the data. The CC-LUT can be implemented using duplicate tags. We note, however, that providing for the CC-LUT can be a large area penalty. The write request must also be forwarded to main memory to update any coherency state held at the home node, the node containing the main memory address[5].

With the second option, *target invalidate* (TI), an invalidate request is sent to the main memory location holding the data. This main memory is located on the *home node*, the back-end store for the data. Each main memory has a directory, the *home directory*, of all the clusters holding a copy of the data line. Main memory updates its directory so that the entry indicates that only the writing node holds a valid copy of the data. Main memory then forwards invalidates to those clusters currently holding copies of the data. Upon receiving the invalidate, the cluster checks the CC-LUT to determine which nodes hold the data. The CC-LUT is updated and invalidates are forwarded to those nodes holding the data[6].

Again, we can make a trade-off of bandwidth for area in both the BI and TI options. The home directory may contain information at the node level instead of the cluster level. The advantage of this option is that there is then no need for each cluster to hold a directory, the CC-LUT, of the addresses being held. However, the

[5] This protocol does not scale well for systems with a large number of nodes. Several groups have looked at methods of scaling coherency information to many nodes, most recently [17].

[6] Most current directory machines use the second-level cache as the CC-LUT.

home directory will be larger, by a factor of the cluster size. Holding information in the home directory at the node granularity instead of the cluster granularity also allows the cluster size to be flexible *after* the time of fabrication – either at compile-time or run-time.

As another trade-off of bandwidth for area, the home directory can maintain information only at the cluster level, and invalidates are sent to *all* nodes in the cluster. Likewise, in the BI case, the CC-LUT is not maintained and invalidates are sent to all clusters on a write, making the number of invalidates that are sent increase by the size of the cluster.

Upon a write request or an invalidate, if the port is already being used by a cluster request, the write request is held in the *write buffer*. The write request has the lowest priority and only gains control of the cache port if the port isn't being used by its own processor's request or a cluster request. If the write request queue is full, a write request has priority over other requests. Figure 3.6 shows the write request protocol.

Read Requests

Figure 3.7 shows the synergistic protocol for a read request. Upon a request, the processor first checks its own cache as well as the write buffer. If the data are in both the local cache and the write buffer, the copy in the write buffer is returned to the processor. The figure shows the priority of the write buffer and the local cache. However, in time, the local cache and the write buffer are checked simultaneously. Upon a hit in either of these locations, the data are returned to the requesting node and the request is complete.

If the processor misses in both its local cache and the write buffer, it then checks the *cluster cache*. Upon a hit in a cache from a neighboring node, the data are returned to the requesting node. At the requesting node, if the port is not available, the data are kept in a buffer queue, the *write buffer*, and updated in the requesting

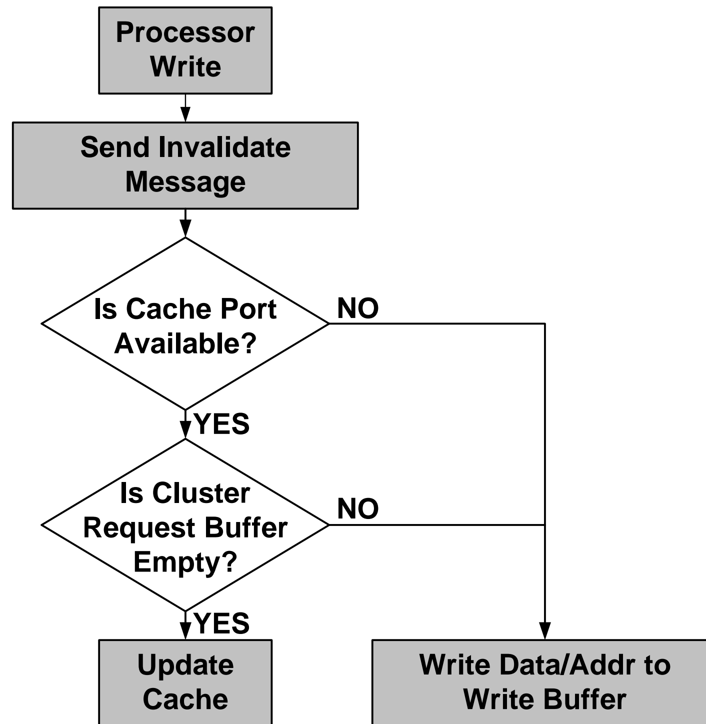

Figure 3.6: Write Request Protocol.

node when the port becomes available. The data in the pending register is updated immediately upon receiving the data.

Only upon a miss in the cluster cache does the processor retrieve the data from main memory. There are two options for requests to main memory: (1) the request is sent simultaneously with the cluster request or (2) the main memory request is sent only after the cluster request returns a miss.

In the first case, the serial delay added by the request to the cluster caches is taken out of the critical path of the memory access time. So, when the request is sent simultaneously to memory, in the worst case, synergistic caches perform as well as independent level-1 caches. And in the best case the data are retrieved from a neighboring node and the memory access delay is on the order of several cycles instead of tens or even hundreds of cycles to access main memory. However, the overhead is

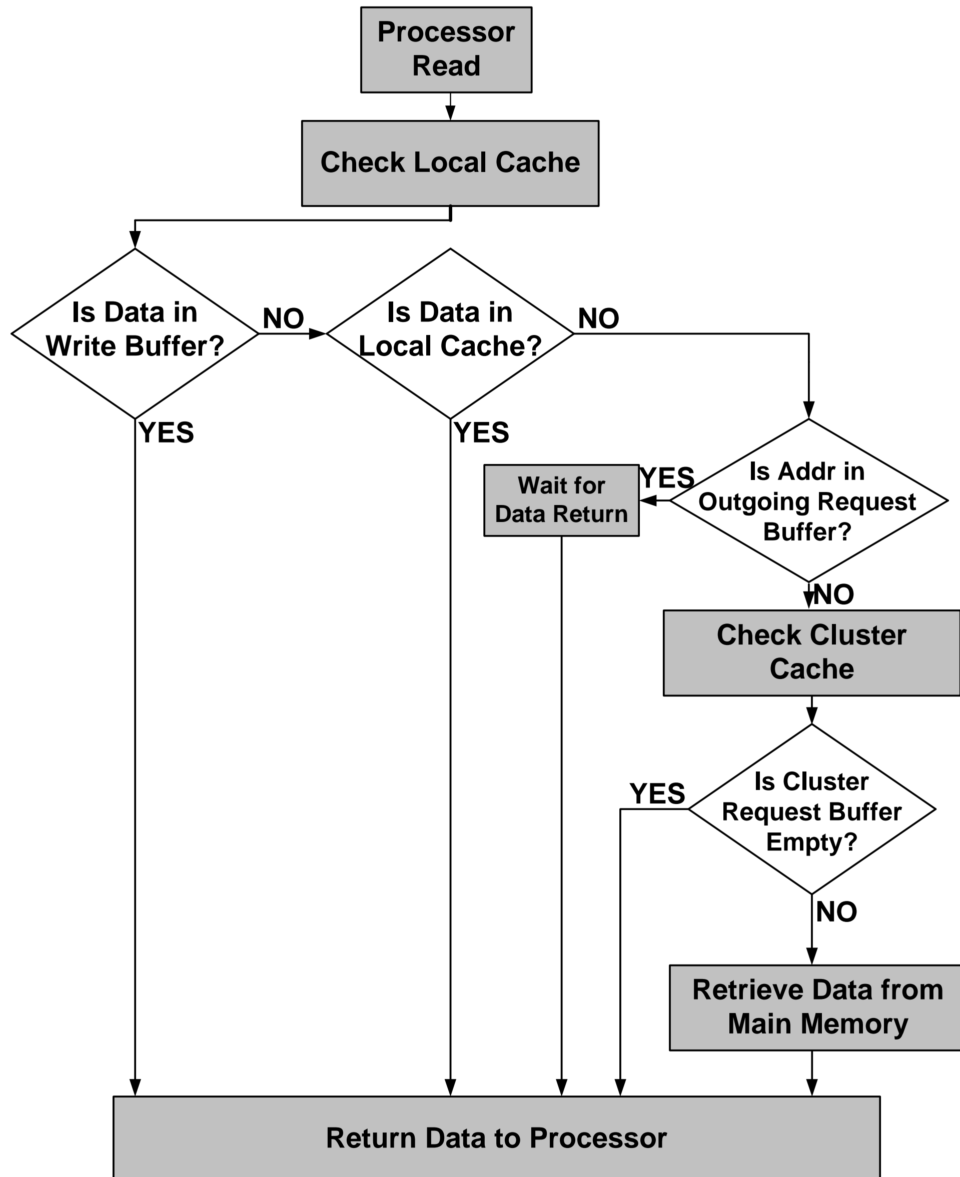

Figure 3.7: Read Request Protocol.

additional network traffic and congestion.

If the request is satisfied by the cluster cache, upon reaching main memory, main memory checks the directory and sees that it is already being held by the cluster and drops the request. If, upon checking the directory, main memory sees that it's *not* held

by the cluster, main memory updates the directory with the new node information and sends the data back to the requesting node. Further discussion of directories and directory protocol is given in the section on coherency below.

The second option is that the main memory request is sent only after all the requests to the cluster caches have missed. In this case, upon a miss, the delay to request from the cluster cache is directly added to the critical path—that of a data miss. However, upon a hit in the cluster cache, the additional request to main memory is not needed, and network traffic is minimized.

In our simulations, we model a system where the main memory request is sent after the local cache miss. This offers three main advantages: First, the delay for missing in the cluster cache is not added to the critical path of data access time. Second, at each neighboring node, when there is a miss in the cache upon a cluster request, no reply needs to be generated by the neighboring node. Also, the requesting node doesn't have to keep track of which nodes have replied to know when to forward the request to main memory. Third, when using a home directory with node-granularity, the directory must be updated anyway. But, upon a hit in the cluster cache, this update of the directory at the home node is kept out of the critical path of memory access time.

Since multiple processors may request data from a given cache, we impose a strict priority or ordering for the incoming requests. In the next section we discuss the priority assigned to each request.

Request Priority

To arbitrate simultaneous requests to the cache, each request type is assigned a priority. The owning processor's read requests have priority for access to its local cache. Cluster requests from other nodes are held in a queue, the *cluster request buffer*, and serviced in the order they arrive when the port is not being used by the owning node.

Table 3.2 shows the priorities of each cache request type. A higher number indicates a higher priority.

Table 3.2: Memory Request Priorities.

Request Type	Source	Request Priority
Read	Owning Node	2
Cluster Read	Cluster Node	1
Write	Owning Node	0
Invalidate	Cluster Node or Other Node	0
Reply	Cluster Node or Main Memory	0

Writes from the owning node as well as invalidates and replies from other nodes have the lowest priority and are kept in the *write buffer* if the cache port isn't available. The write buffer obtains access to a cache only after read requests from the owning node and cluster requests from other nodes are serviced. Table 3.2 shows the priorities of each of the actions required of each cache. A higher number indicates a higher priority.

If the write buffer holding waiting requests fills, it momentarily obtains priority over the cache port. However, if the cluster request buffer fills, the oldest requests are simply dropped by sending a negative reply back to the requesting node. In the next section we briefly discuss consistency and coherency in the synergistic caching system.

3.3.5 Coherency and Consistency

In this thesis we treat coherency as an orthogonal issue to data sharing. The coherency information will need to be updated regardless of which location the data is retrieved from. In the simulations given in Chapter 4, coherency delays are not included in either the independent-L1 cache simulations or the synergistic system as that overhead will be similar for either system. However, in the system architecture we propose that

caches be kept coherent with a write-invalidate over the network. Similar to other coherency policies, a directory indicating which nodes hold copies of a memory line is kept at each home node.

As mentioned previously, in the synergistic caching system, the coherency information might even be smaller than in independent node systems. The granularity of information held in the directory may be kept cluster level instead of at the node level. However, if information at the home directory is at the granularity of the cluster, each cluster must also have a cluster cache look-up table (CC-LUT).

There have been several methods for minimizing the area of directory structures as well as improving time and resources needed for coherency. Additionally, since the directory traffic overhead is required for a synergistic or non-synergistic caching system, this overhead is normalized out of the simulation. As this is not the focus of this research, we refer interested readers to other work focusing on coherency in [13][1].

We use a relaxed-consistency model. Synchronization instructions called by a single thread are guaranteed to execute in sequential order. However, instructions between synchronization calls may execute out-of-order. Read after Write hazards as well as write after write hazards are avoided. Read after read hazards must be controlled by the application programmer.

3.3.6 Summary

In this section we described the basic protocols, consistency and coherency models, and the additional hardware required for synergistic caching. In the next section we discuss variations on the baseline synergistic caching system described above.

3.4 Synergistic Caching Variations

In this section we discuss the basic question of when and how to copy data. The fundamental trade-off is overall capacity versus latency of access. We explore three duplication strategies: *Beg*, *Borrow*, and *Steal*. We introduce each duplication strategy, describe the differences, benefits, and drawbacks of each strategy. In Chapter 4 we explore the performance of each strategy across different applications and cache sizes.

3.4.1 Duplication Policies: Beg, Borrow, Steal

Each of the modes—*beg, borrow*, and *steal*—defines whether data are duplicated at the requesting node upon a hit in the cluster cache. Also, depending on the duplication mode, the state of the data in the supplying cache may be affected. In the case of the *steal* mode, the state of the data in the supplying cache is also changed.

Figure 3.8 shows cartoons of each of the duplication modes. This figure shows the (a) *beg* mode, (b) the *borrow* mode, and (c) the *steal* mode. The solid lines represent requests for data, and the dashed lines represent data replies.

In the *beg* duplication mode, the processor requests the data from a neighboring node. Upon a hit in the neighboring node, the processor duplicates the data in its own cache. In the *borrow* duplication mode, the processor requests the data from a neighboring node. Upon a hit, the processor uses the data but doesn't duplicate it in its own cache. In the *steal* duplication mode, the processor requests the data from a neighboring node. Upon a hit, the processor invalidates the data in the supplying node and copies it into its own node.

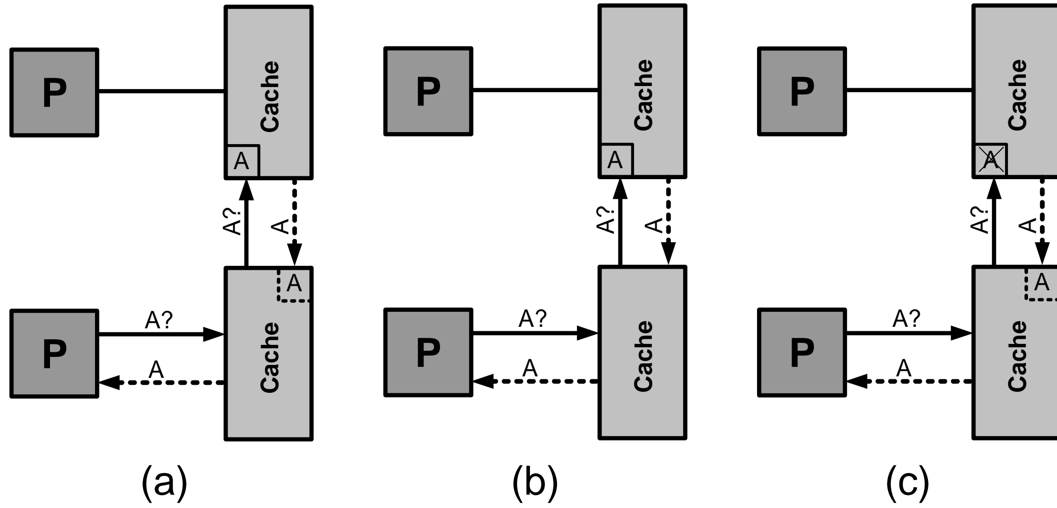

Figure 3.8: Duplication Modes.

3.4.2 Capacity

The performance of a caching system is greatly affected by the overall cache capacity of the system. In this and the following section we compare the effective capacity of each duplication mode and examine the resulting effects on performance. In discussing overall capacity limitations, we define *cluster cache capacity* as the total number of unique bits stored in a cluster. The worst-case cluster cache capacity occurs when the maximum number of bits are duplicated in the system, and the best-case cluster cache capacity has the minimum number of duplicated bits.

Figure 3.3 shows the maximum and minimum cluster cache capacities for the *beg*, *borrow*, and *steal* variations. In the figure, CS denotes the cluster size, and $L1_size$ is the size of the the local cache.

Table 3.3: Cluster Cache Capacities.

Duplication Mode	Best-Case Capacity	Worst-Case Capacity
Beg	$CS \times L1_size$	$L1_size$
Borrow	$CS \times L1_size$	$CS \times L1_size$
Steal	$CS \times L1_size$	$CS \times L1_size$

As shown in Table 3.3, with the beg duplication mode, the worst case is when all neighboring caches contain the same data. In this case, querying from neighboring nodes will be of no benefit because the neighboring nodes hold no unique data. But if the data are in fact used by each of the node and the application is not capacity-limited, this duplication mode is optimal. The requested data can be repeatedly accessed directly from its own cache instead of across the network in a neighboring cache.

However, if the application is capacity-limited—i.e. needed data are evicted from the cache due to capacity constraints—the worst-case capacity of the beg mode is sub-optimal. Instead of storing unique data in each of the caches and accessing a larger overall data set with possibly small penalties for accessing neighboring nodes, the capacity is filled up by redundant, duplicate data and is essentially wasted.

Thus, the *beg* mode minimizes the access time to frequently accessed shared data when the application is not capacity-limited, but it also has the minimal worst-case cluster cache capacity.

On the other hand, with the *borrow* and *steal* modes of duplication, the best-case and worst-case capacities are the same. Using these modes, each of the cluster caches will always hold unique data from the other caches in the cluster. However, this effective increase in capacity over the beg mode may not be beneficial. In fact, it could have deleterious effects if frequently accessed data of one node are held in a neighboring node's cache. Upon frequent accesses of this data, the small penalty of accessing the data from the neighboring cache, instead of from its own cache, adds up and may overwhelm the benefits of added overall capacity. We examine these effects in detail in Chapter 4.

The *borrow* and *steal* duplication modes maximize cluster cache capacity, but frequently-accessed shared data held in neighboring caches incur incrementally large penalties.

We propose that the optimal duplication mode depends on the data footprint of the application and on how capacity-limited the application is.

3.4.3 Performance Trade-offs

As described above, the *beg* option minimizes the access time to frequently accessed shared data, but it also has the minimal worst-case cluster cache capacity. And the *borrow* and *steal* options allow us to maximize the worst-case cluster-cache capacity since there are no duplicate data among cluster caches and the cluster cache capacity is maximized.

Particularly in the case of *steal*, shared data incur a maximal penalty in delay if the node from which the data are stolen hasn't finished accessing the stolen data. The data essentially ping-pongs between the cluster caches. The worst-case scenario is where the shared data are alternately requested and invalidated by neighboring nodes upon alternate requests by each node. In this case, neither node reaps the benefit of hitting in its own cache, and each node pays the penalty of accessing the data from the cluster cache upon each request. Note that this may still be better than going to main memory to retrieve the data, but the incremental penalty of accessing a neighboring node upon every request adds up and may overwhelm the benefit of sharing.

In the next chapter, we explore which duplication option performs best across a set of applications. We propose that there isn't a single optimal duplication mode. Depending on the application, and particularly the data footprint of the application, the optimal duplication mode varies.

3.4.4 Duplication Hardware

For each of the duplication options described, there is little additional required hardware. A flag in the cache controller sets which duplication mode is being used. Upon detecting which mode is used, the data are forwarded to the local cache or simply used by the requesting processor. As mentioned, if we were to implement feedback in the system - to allow it to switch duplication modes at run time, it would need a small amount of additional hardware. We discuss these additions in this section.

Reconfigurability

There are three options for duplication mode definition: *static*, *compile-time*, and *run-time*. With a static definition, the duplication mode is hardwired into the system and is not configurable. With this option, after the system is built, the duplication mode cannot be changed.

With compile-time duplication, the duplication mode is determined by a directive from the application or as a result of compile-time analysis of the application. This requires either previous knowledge of the application characteristics by the programmer or analysis of instruction blocks by the compiler. A caveat for the last option is that if the compiler—or the programmer—guessed wrong, the application will complete its execution in the slower, non-optimized mode.

The last option, run-time duplication, allows the hardware to adapt to different duplication modes depending on the performance of the application. This would allow an application to change to a *borrow* or *steal* duplication mode when it is capacity-limited. Likewise, an application that is not capacity-limited would be able to use the *beg* duplication mode to minimize the time to access repeatedly requested data.

Duplication modes can also be selected for different granularities. For example, the duplication mode can be set for the entire code, for a code segment or for specified

cache lines.

Run-time configurability and reconfigurability require feedback. The system can track the number of hits in the cache for a given time period, and at the end of a measurement period, if the number of hits is below a set threshold, the mode would change from *beg* to *borrow*. The threshold itself can be set by the user or the compiler or using feedback itself. If the program detects better performance using a given mode but due to the threshold it keeps jittering between modes, it can reset the threshold to correlate to the better performing mode. This feedback loop only affects the shared data in the system. The duplication of non-shared data is left unaffected.

In order to detect between shared and non-shared data, the network interface reads a tag that indicates the source of the data, main memory or a cluster cache. Then, depending on the mode, after the processor consumes the data, it may or may not duplicate it in its local cache. If the duplication mode is *borrow*, upon detecting that the source of the data is a neighboring cache, instead of main memory, the data are sent directly to the processor and not duplicated in the cache. However, in the case of *beg* and *steal*, a copy of the data are made in the cache. At the supplying node, if the duplication mode is *steal*, after supplying the data, the cache invalidates that data line in its cache.

3.5 Area

In analyzing the synergistic caching system, we look at both the performance of the system as well as the on-chip area occupied by the system. We use the Cacti 3.0 cache access and timing model to predict both the access time and overall cache area used by each system[26].

Table 3.4 shows the approximate chip areas for each of the cache sizes used in our analysis. We use the Cacti 3.0 system to generate estimated values using a 65 nm

process. The table shows both the overall areas used by each of the systems and the access times for each cache size.

Table 3.4: Cache Sizes.

Cache Size	Delay (processor c.c.)	Area (mm^2)
1 KB	1	0.103
2 KB	2	0.118
4 KB	2	0.123
8 KB	2	0.179
16 KB	2	0.275

For each of the cache sizes given, there is one read/write port and one bank. In the next section we analyze the potential benefits of the synergistic caching system. We propose the conditions in which the system will perform well. In the next chapter, Chapter 4, we test these hypotheses by looking at performance data of the simulated machine.

3.6 Benefits of Synergistic Caching

We've explored the basic mechanics of the synergistic caching system and looked at several variations in duplication modes. We now examine the cases in which this system will be beneficial in increasing application performance.

The synergistic caching system performs particularly well in the following three cases: when there are large amounts of sharing in an application, when an application is capacity-limited, and when an application has low latency tolerance. We discuss each scenario in the following sections.

3.6.1 Sharing

The characteristics encouraging communication between nodes are high bandwidth and low latency of CMPs and sharing in multi-threaded applications. In this section

we focus on the last characteristic, sharing in multi-threaded applications.

Synergistic caching takes advantage of the sharing present in many multi-threaded applications. This sharing can be spread across time–throughout different iterations. The sharing can also be spread across physical space—distributed across different processors on the chip.

Multi-threading breaks up the control of an application into multiple threads. In Figure 3.9 we see a single-threaded application on the left of the figure. The single-threaded application has a single list of instructions to execute. On the right side of Figure 3.9 we see a multi-threaded version of the same application where the instructions are broken up into two threads of control.

The two threads shown in the figure, Thread 1 and Thread 2, have shared data, data that they both use. In the case shown, VAR1 and VAR2 are shared by both threads. Likewise, both threads have private data, data that are only accessed by a single thread. Depending on where the data lie in the cache line, the private data accesses may also look like shared accesses. This is called *false sharing*.

Using a multiprocessor platform, Thread 1 and Thread 2 may be run on separate processors. Since they share data, if there is a communication path between processors 1 and 2, the sharing of this data will be facilitated. Synergistic caching does exactly that—it takes advantage of the on-chip multiprocessor environment to enable easy sharing of data between neighboring nodes.

Synergistic caching takes advantage of the large amounts of sharing present in many multi-threaded applications. This sharing can be spread across time–throughout different iterations. The sharing can also be spread across physical space—distributed across different processors on the chip. We characterize shared and private data further as described in Table 3.5.

We measure the number of shared and private data accesses in each category by simulating each application running with an ideal memory system, where each

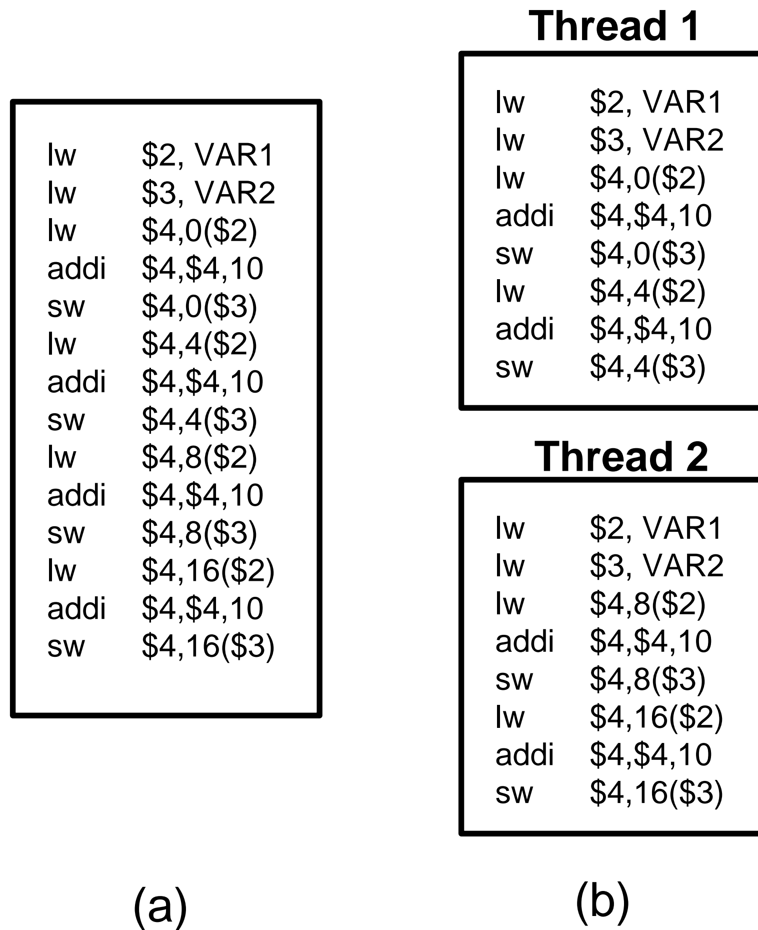

Figure 3.9: Multi-threading.

memory access takes a single clock cycle. A thread is considered concurrent with another thread if any part of its execution overlaps with the execution of the other thread. Conversely, two threads are considered non-concurrent if their executions don't overlap.

Synergistic caching enables easy access to all three types of shared data—concurrent, non-concurrent, and both—by closely coupling the data accesses of neighboring processors. In the next section we talk more about capacity limitations and their effects on application performance.

Table 3.5: Sharing Categories.

Category	Type	Description
Private	Single-Access	Memory is accessed by a single thread and is accessed only once.
	Multiple-Access	Memory is accessed by a single thread, but it is accessed multiple times by that thread.
Shared	Concurrent	Memory is accessed by multiple threads that are running concurrently.
	Non-Concurrent	Memory is accessed by multiple non-concurrent threads.
	Both	Memory is accessed by threads that are running concurrently as well as threads that are running non-concurrently.

3.6.2 Capacity-limits

As previously mentioned, in a synergistic caching system, if an application is capacity-limited, the neighboring cluster caches act to effectively expand the cache capacity. Data that are evicted due to capacity constraints may still be held in neighboring caches. So, instead of the processor going to main memory to retrieve the data after they have been evicted, it can pay a much smaller delay penalty and retrieve the data from a neighboring cache.

In the next section we discuss the third characteristic of an application that determines its performance when running on a synergistic caching system: latency tolerance.

3.6.3 Latency Tolerance

The higher the latency tolerance of an application, the less sensitive it is to changes in the memory system. Two of the most common reasons for high latency tolerance in an application are: the application has a large load-use distance, or many threads are running concurrently on a single node.

A large load-use distance means that a data item is requested many cycles before it is actually used by the processor. Figure 3.10 shows two applications. Application 1, on the left side of the figure has a load-use distance of 1 processor clock cycle for register \$4. Application 2, on the right side, has a load-use distance of 10 processor clock cycles for register \$4. The second application could tolerate a 10-cycle delay in the memory system without the access delay affecting overall execution time. However, any delay in the memory system of the system running the first application will be seen directly in the execution time of that application.

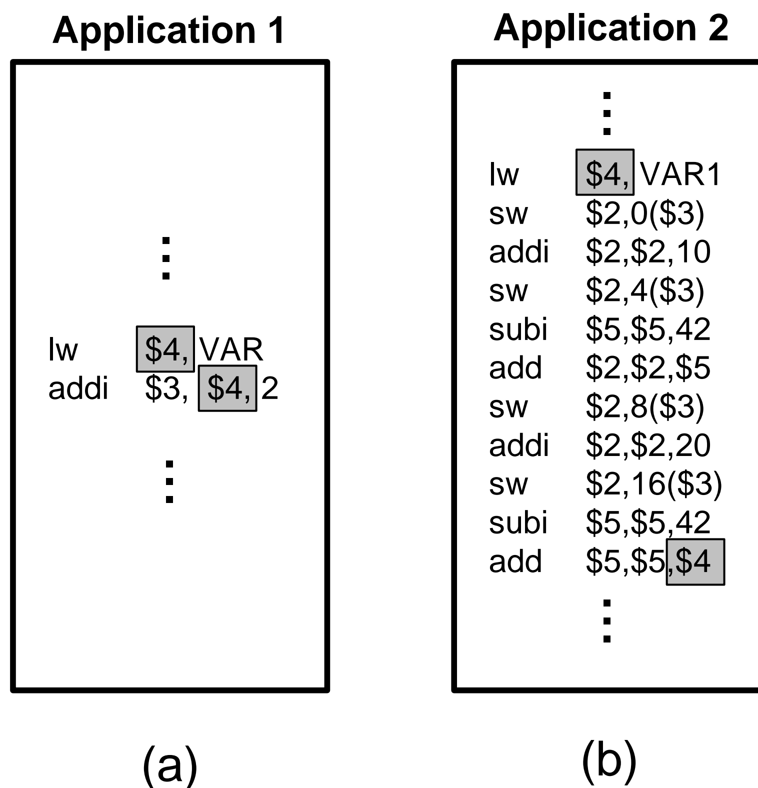

Figure 3.10: Load-use distance.

The second major cause of latency tolerance in applications is multi-threading (MT), many threads running simultaneously on a single node. With MT, while one thread stalls for memory, the other threads can be using the processor clock cycles to

do useful work. Figure 3.11 shows an example of two threads running simultaneously on a single processor. The latency of thread 1 waiting for data A is effectively hidden by thread 2 being able to do useful work. Even though the execution time of thread 1 doesn't improved, the overall application is able to progress.

	Thread 1	Thread 2
Cycle 0	lw $4,VAR	
Cycle 1	addi $3,$4,2	
Cycle 2		lw $5,VAR1
Cycle 3		sw $2,0($3)
Cycle 4		addi $2,$2,10
Cycle 5		sw $2,4($3)
Cycle 6		subi $5,$5,42
Cycle 7		add $2,$2,$5
Cycle 8		sw $2,8($3)
Cycle 9		addi $2,$2,20
Cycle 10	data $4 ready	sw $2,16($3)
Cycle 11		subi $5,$5,42
Cycle 12		add $5,$5,$4
Cycle 13	addi $3,$4,2	
Cycle 14	subi $3,$3,1	
⋮	⋮	⋮

Figure 3.11: Multi-threading.

At cycle 0, Thread 1 loads data into register 4 and the data is not available in the local cache. However, the Thread doesn't stall until after cycle 1, when the data

is used. At this point the processor switches contexts to Thread 2[7]. Thread 2 runs until it stalls at cycle 12, when it needs to use data loaded into register 5, but this data is still being retrieved. At this point, Thread 1 can continue running. In fact, Thread 1 was ready to run at cycle 11, but it needed to wait for a context switch[8].

There are several caveats with multi-threading. The first is that to switch contexts from one thread to another, the state of the stalling thread must be stored and the state of the new thread must be restored to the active state of the processor. The state can be as minimal as restoring the registers, the stack pointer and the program counter. In multi-threaded systems, there are often multiple register sets, making switching contexts quick. A normal (non-multi-threaded) processor can utilize multi-threading by saving and restoring registers at the time of context switching.

The second caveat with MT is that if each thread is not working on overlapping data, switching contexts from one thread to another may pollute the cache. That is, one thread may evict the data needed by another thread. This is called *cache interference*. Also, not all applications lend themselves to be written using MT. Interested readers are referred to [16][30] for further inquiry.

Applications that have high latency tolerance benefit little, if at all, from enhancements in the memory system, such as synergistic caching. These applications can already tolerate large memory delays. Thus, just as they hide latencies of poor performing memory system, they also hide improvements in the memory system. However, there still may be a benefit from decreased bandwidth demand.

On the other hand, applications that have low latency tolerance or small latency tolerance, synergistic caching can be beneficial. With low latency tolerance, the application execution time depends on the memory access time; so lower memory

[7]The figure doesn't show the time needed for context switching.

[8]There are many methods for determining when a node switches contexts from one thread to another. Some options are: switching threads upon an interrupt, switching threads after a set number of cycles, or, as shown in the example, switching contexts only after the running thread stalls.

access times lead directly to lower execution times.

3.7 Summary

In this chapter, we introduced the synergistic caching system that allows neighboring caches to share data at the first-level cache. We defined both the architecture and the protocols governing the system. We also defined three duplication modes: *beg*, *borrow*, and *steal*. We proposed that synergistic caching performs well when there are large amounts of sharing in the application, when the application is capacity-limited, and when the application shows low latency tolerance. In the next chapter we introduce the experimental framework and applications we used to test these hypotheses.

Chapter 4

Performance Results

In this chapter we describe the experimental framework for testing the synergistic caching system. We describe the simulator we designed and the set of benchmarks we simulated to explore the behavior of synergistic caching. We examine the characteristics of the applications that determine performance in a synergistic caching environment and analyze the simulated performance results.

4.1 Experimental Set-Up

To test the synergistic caching system, we wrote a cycle-accurate simulator that simulates a 64-node single-chip multiprocessor built in 65nm technology running at 2.5 GHz. Figure 4.1 depicts the experimental set-up we used.

Each node contains a processor, an L1 cache, and a portion of the main memory. We model main memory accesses with a uniform delay of 50 clock cycles[1]. We scale

[1] In the actual system, a portion of main memory is held on each node. 50 clock cycles is used in the model as the *average* time to retrieve a data line from memory. This average access time is configurable in the model. In simulations done for varying average main memory access times, the trends were the same as those that are presented in this chapter. Simulations were also done sending requests across the network to the node holding each memory location. Using the average time is an approximation used to speed up simulation time.

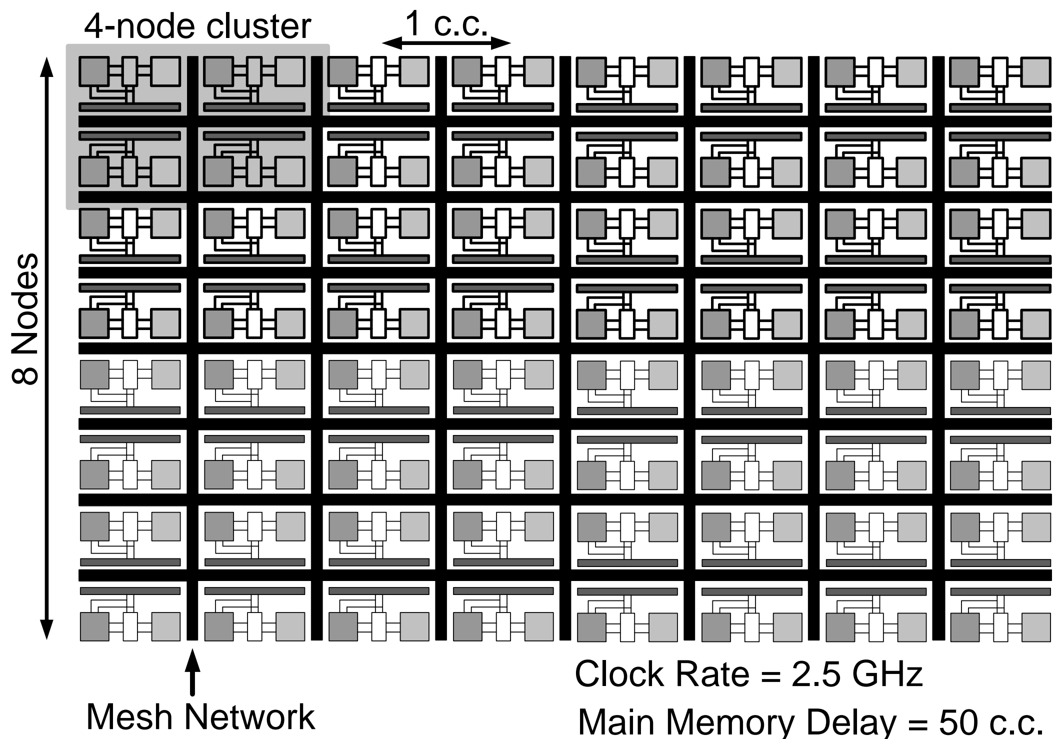

Figure 4.1: Synergistic Caching Experimental Set-up.

the MIPS64 core for 65nm technology such that each processor core takes $1mm^2$ of the total chip area. We model a relaxed consistency distributed shared memory (DSM) system with non-blocking accesses. Each node has an L1 cache which is 2-way set associative, has a line size of 64 bytes and employs least recently used (LRU) replacement. We generate the results reported in the following sections by simulating a 4-node cluster[2] of the 64-node system.

The delay to neighboring nodes is modeled as a mesh network without contention. The delay per hop is one processor clock cycle. Thus, the delay to reach data in a

[2]Experiments were made on a range of cluster sizes. These experiments showed that the cluster size of four was optimal. For smaller cluster sizes (i.e. a cluster size of 2), the cost of communication was about the same as the 4-node cluster (about 1 hop to the neighboring node), but the benefits of synergistic caching showed only marginal performance improvement. For larger cluster sizes (8, 16, etc.) the cost of communication (added congestion and delay) overwhelmed the benefits of additional cache capacity and the performance decreased.

neighboring cache is two cycles plus the cache access delay. The total access time of the L2 cache is the wire delay to the L2 cache controller (1 processor clock cycle for the round-trip delay) plus the access time of the L2 cache. All local L1 accesses incur a delay determined by the cache size. See Table 3.4 for timing details.

4.2 Applications

The experiments detailed in the following section were run on a set of five fine-grain multi-threaded applications described in Table 4.1. The programs range from an image processing application–*median_filter*, and n-body simulations–*nbody* and *barnes-hut*, to a graphics application–*trace*, and a CAD application–the *lotus* wire-router.

We simulate the kernel, the parallel portion, of each fine-grain multi-threaded application running on a MIPS64 core with single-instruction in-order execution.

Table 4.1: Benchmarks.

Application	Description	Memory Footprint (Bytes)
barnes-hut	An O($nlogn$) n-body simulation.	8875904
medianfilter (mf)	Performs a median filter of an nxn array of pixels.	1970304
lotus	Performs a lotus wire-route of a set number of wires on an nxm grid.	148928
nbody	An n^2 n-body simulation.	33088
trace	Produces an output image using ray tracing.	430656

We focus on the three key characteristics of these applications that affect overall performance when running on a synergistic caching platform, namely: the percentage of memory stall cycles, the overall amount of shared data, and the amount of shared

data available on neighboring nodes. In the next sections we examine each of these application characteristics in detail.

4.2.1 Memory Stalling

The first characteristic affecting overall memory performance is the percentage of execution cycles an application spends stalling for data instead of doing useful work. If an application spends few cycles waiting for memory, there is very little margin of improvement for the memory system. On the other hand, large amounts of stalling, while sub-optimal for the application performance, give the opportunity for large improvements in the memory system.

An application may spend few cycles waiting for memory for several reasons. The most prominent of these are that the application is dominated by computation, because the L1 cache services most of the accesses, or because the application inherently has high latency tolerance. In each of these three scenarios, delays in the memory system are only a small percentage of overall execution time and the application is, thus, mostly impervious to changes in the memory system.

Figure 4.2 shows the percentage of memory stall cycles for each application running on a traditional caching system - without synergy. The L1 cache sizes are being swept from 1KB to 8KB.

These statistics show the margin of improvement each application offers by enhancing the memory system. Figure 4.2 shows that all applications spend some amount of processor cycles stalling on memory. Across applications, 15% to almost 70% of processor cycles are spent waiting for memory and are wasted. The greatest margin for improvement is at smaller cache sizes.

We note that the *barnes-hut* application shows the least room for improvement across all cache sizes with the percentage of cycles spent stalling for data are from only 14% to 19% of execution cycles. The other applications average just around 40

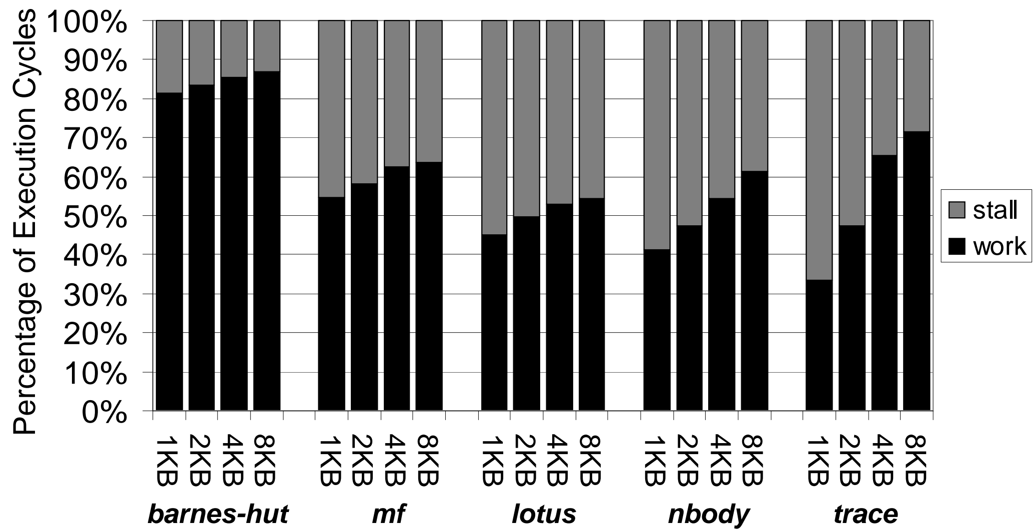

Figure 4.2: Percentage of Stall Cycles.

to 50% of execution cycles being stall cycles. We will come back to this point when we discuss this application's overall performance improvement using the synergistic caching system.

The applications show that there is a margin for improvement in the memory system. They are, indeed, being affected by the processor-memory gap and enhancing the memory system by using synergistic caching could improve the performance of the applications and bridge the processor-memory gap.

However, before we determine how much synergistic caching will improve the system, we must first determine if there is shared data that can be exploited by the synergistic caching system. We explore the amounts of shared data available in each application in the following section.

4.2.2 Shared Data

In the previous chapter, we showed how synergistic caching takes advantage of shared data available in multi-threaded applications. But the underlying question remains:

how much of the data accessed by an application is shared? The more shared data that is available, the greater the potential advantage of synergistic caching.

We examine each of the applications in the benchmark suite for the amount of shared data present. In Figure 4.3 we show each application with the overall percentage of shared memory accesses and private accesses. The left column for each application shows the percentage of memory lines that are accessed. The right column shows the percentage of total memory accesses. These numbers are independent of the underlying architecture.

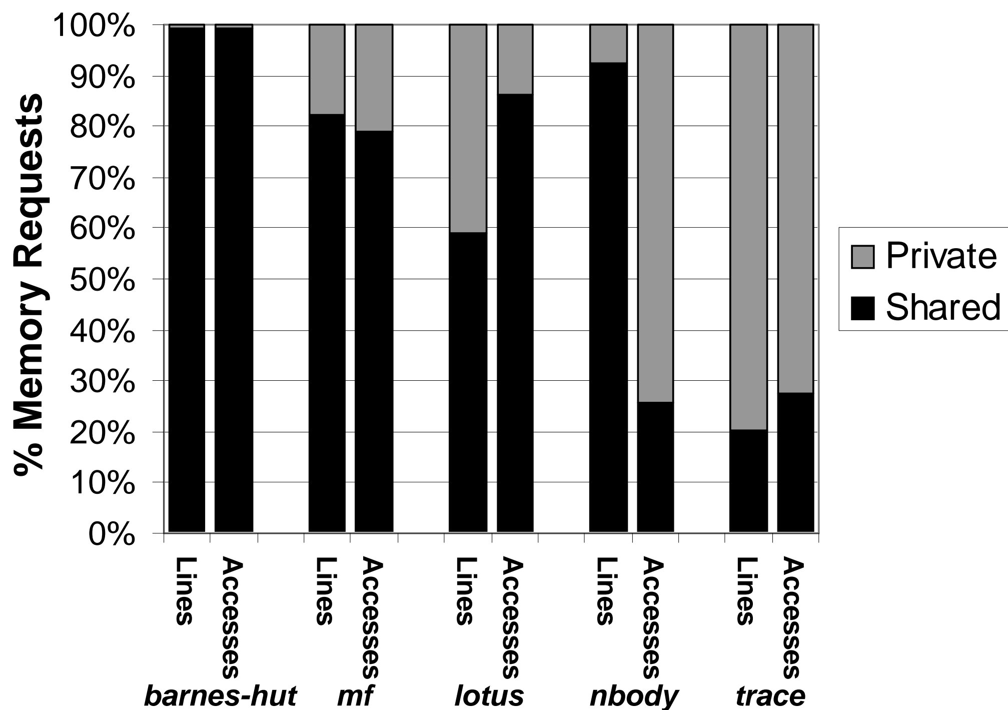

Figure 4.3: Shared Data.

The graph shows significant amounts of shared data in all of the applications. Across the applications, as shown in Figure 4.3 from 15-99% of data accesses are to shared data. In the figure we also show a graph of the shared and private address lines. Since the ratio of shared and private accesses remains about the same as in

Figure 4.3, we see that the number of repeat accesses to shared data and private data are about the same. That is, a shared address line is about as likely to have repeated requests from a processor as a private address line.

Figure 4.4 shows the shared and private data in greater detail: It breaks up private data into single-use and multiple use, and it breaks up shared data into concurrently-used, non-concurrently used, and both concurrently and non-concurrently used.

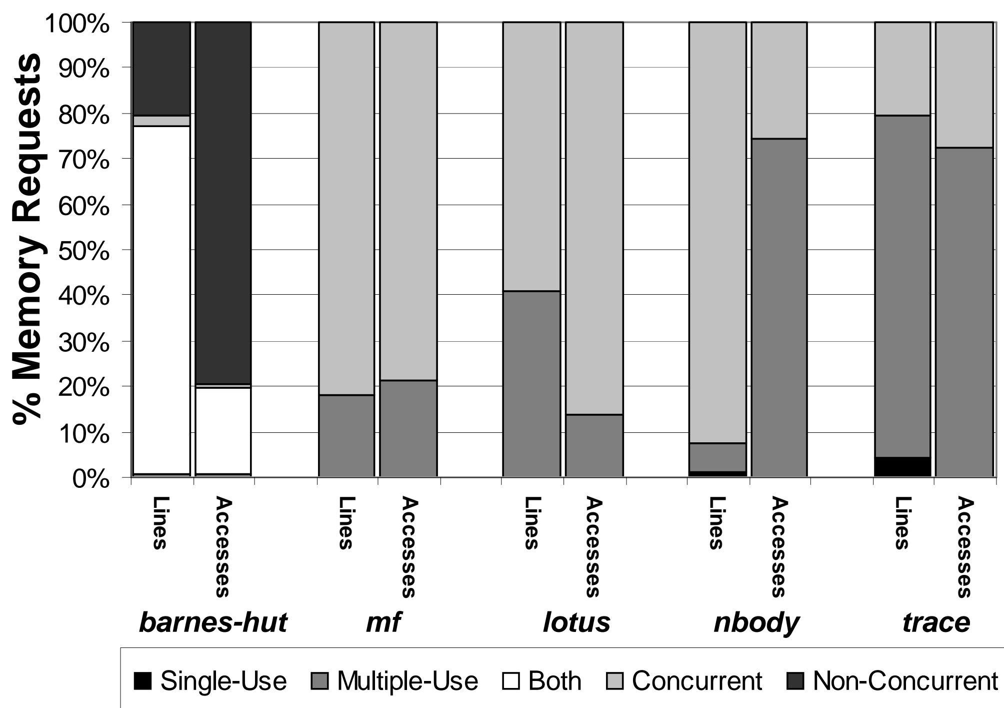

Figure 4.4: Shared Data Details.

The graph show several trends. First, the amount of single-use data is almost non-existent. Second, we notice that most data accesses are either multiple-use private or concurrent shared data. Single-use private data and non-concurrent shared data are a small percentage of total accesses. But *barnes-hut* is an anomaly with most of its data being either non-concurrently shared or both non-concurrently and concurrently shared. In fact, 80% of the data accesses fall in the former category. When we examine

execution results in the following sections, we'll refer back to this characteristic.

Figure 4.5 shows the shared and private data for reads and writes. Again the private and shared accesses are broken down as in Figure 4.4.

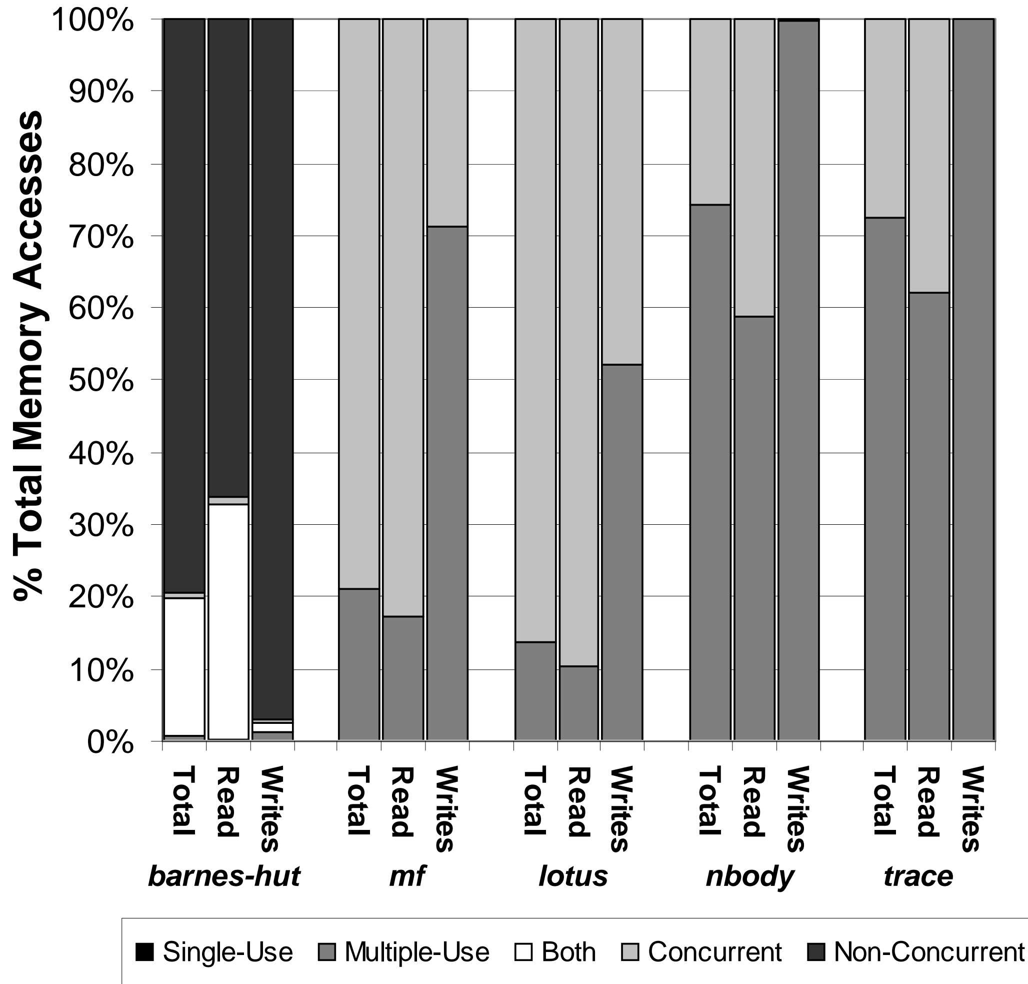

Figure 4.5: Shared Reads/Writes.

The two main observations from this graph are that the percentage of shared writes as compared to private writes is lower, and often significantly lower, than the percentage of shared reads for all of the applications except *barnes-hut*. Particularly in the *trace* and *nbody* applications, almost all—100% and 99.8%, respectively—of

the write accesses are private.

With most write data being accessed by a single thread, this means that synergistic caching speeds up the initial access to a data line. But since most writes are private, it is unlikely to be invalidated later by another thread. Thus, after the first retrieval from the neighboring node, the only reason for a data line to be re-fetched from a neighboring node is if it is evicted from the cache because of capacity constraints. So, in a system with few capacity misses, synergistic caching only ameliorates the "start-up costs" of accessing data. In other words, the first time shared data are accessed, they may be retrieved from a neighboring node. However, in a system with little shared write data and few capacity misses, after the first retrieval, the data are then retrieved from the processor's own cache.

In this section we showed that there are large amounts of data sharing across applications. Across the applications we tested, from 15 to 99% of data accesses are to shared data.

4.2.3 Retrieving Shared Data

In the previous two sections, we showed that the memory system is a bottleneck for all applications and that large amounts of shared data exist across all applications. The final question is then: of this shared data, how much of it is available for use by neighboring nodes to decrease the overall execution time?

This question motivates the remainder of our analysis in this chapter. In the next section we analyze the performance of synergistic caching across several cache sizes. We analyze the performance results for the baseline synergistic caching system and compare them to the independent L1 caching system performance. We also analyze performance across all three duplication modes–*beg*, *borrow*, and *steal*.

4.3 Performance Results

In this section we explore how much synergistic caching improves performance by enabling the available shared data shown in Figure 4.3 to be easily shared within a cluster.

We examine how well our expectations hold for the performance of the synergistic caching system. Specifically, we examine the system to see if it performs well under the hypotheses specified in Chapter 3, namely that the system out-pperforms the independent L1 systems[3] in the following three cases: when there are large amounts of across-chip sharing in the application, when the application is capacity-limited, and when the application shows low latency tolerance.

We first explore the base results for the benchmarks across a range of cache sizes. We compare the synergistic caching results to a system with independent L1 caches. In the baseline system, a *beg* duplication mode is used. We explore the performance as defined by miss rates, average memory access times, and execution times. We then look at how each application performs under each of the duplication modes of *beg*, *borrow*, and *steal*.

4.3.1 Miss Rates

We examine the hit and miss rates of each of the applications across a range of cache sizes. Figure 5.4 shows the hit rate of each application using an independent L1 caching system. We compare the miss rates of the synergistic caching system and the independent L1 caching system in Figure 4.6. This figure shows the *overall miss rates* for each system. The overall miss rate describes the percentage of memory accesses that are serviced by main memory. Memory hits can be serviced by the L1 cache or by the cluster cache.

[3]In Chapter 5 we compare the synergistic caching system with a shared L2 system.

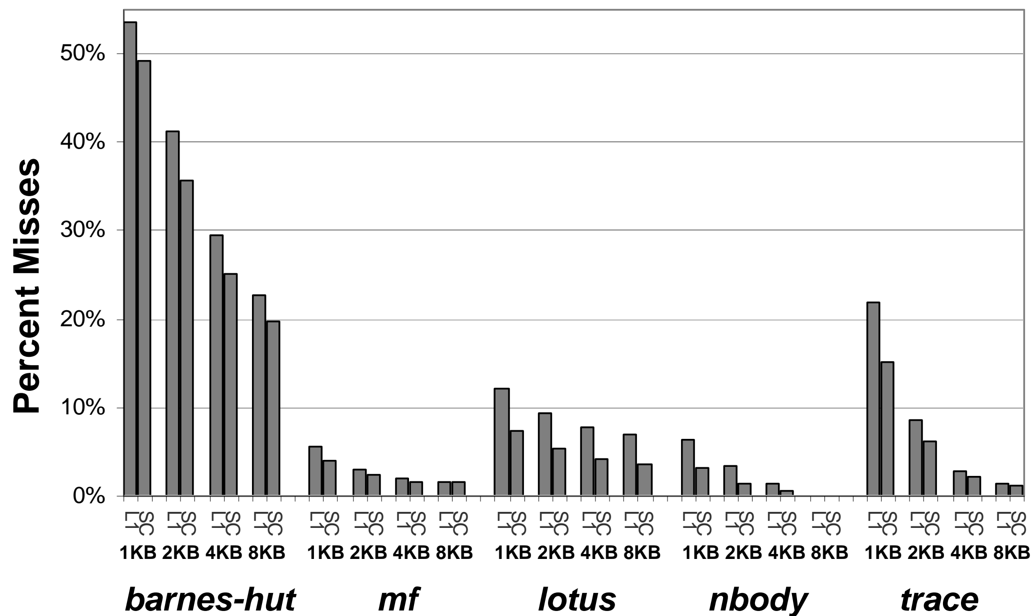

Figure 4.6: Miss Rates for the L1 and Synergistic Caching Systems.

The largest margins of difference between the independent-L1 system and the synergistic caching system are at the points on the L1 miss rate curve with the steepest gradient. All of the applications show significant improvement using the synergistic caching system when the miss rate is high. However, as the L1 hit rate curves approach 0%, the difference between the two overall miss rates also becomes minimal. We look at cache sizes ranging from 1KB to 8KB because that is where, for these applications, the miss rate characteristics show the largest variation.

At the 8KB L1 cache size, only *barnes-hut* and *lotus* show significant differences in the miss rates of the two systems. At this cache size, the other three applications are already at a very low miss rate, less than 2%. There is very little room for improvement. But, using the synergistic caching system, *barnes-hut* and *lotus* still show significant improvements in miss rates at the 8KB cache level—15 and 98% improvements in miss rate over the independent-L1 system, respectively.

In Figure 4.7 we look in detail at where the data accesses are hitting in the memory system. The figure shows the details of which level of the memory system serviced each access. The levels are the first-level cache, and the cluster-cache in the synergistic caching system. The number of memory accesses serviced by main memory are also shown in light gray on the graph. These levels in the memory hierarchy are denoted as L1, SC, and MM, respectively, in the graph.

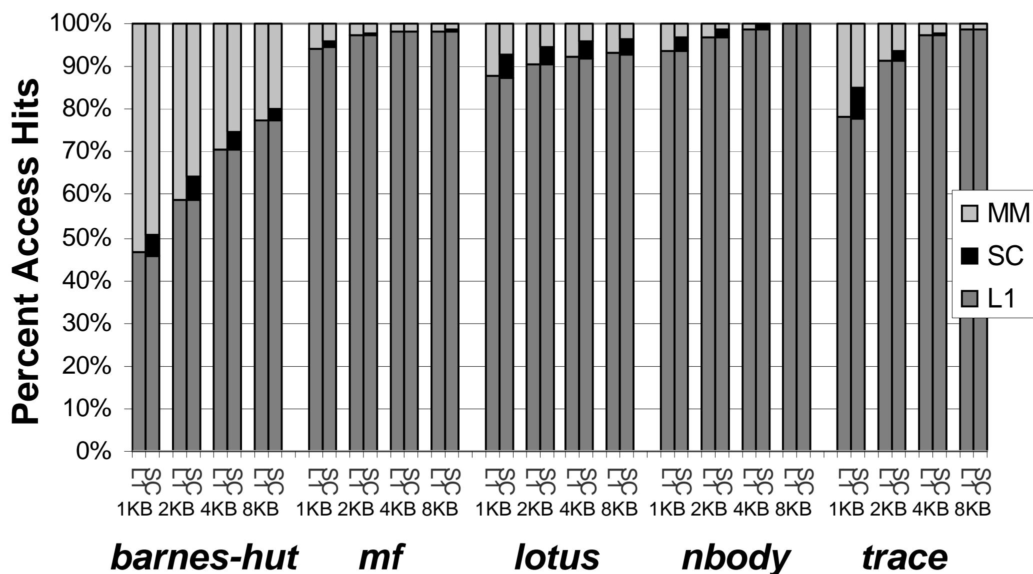

Figure 4.7: Hit Rate Details.

The left bar for each cache size shows the performance of the independent-L1 caching system. The right bar shows the synergistic caching system performance.

Across all cache sizes and applications, the figure shows up to 8% of misses in the L1 cache are hitting in neighboring nodes. In some cases—for example with *lotus* at the 4KB and 8KB cache levels—this means that more than 50% of the misses in the L1 system, that would have been serviced by main memory, are instead serviced by the cluster cache. Consequently, the number of memory stall cycles are reduced by a comparable percentage, as we will show later in this chapter in Figure 4.14.

The first trend in the hit rate data is that synergistic caching is indeed most effective for cases where the application is capacity-limited—for small caches and low L1-cache hit rates. This trend is evident across all of the applications for small cache sizes.

This effect can also be seen in *barnes-hut* and *lotus* for all cache sizes due to their relatively low hit rates across the board. However, as the hit rate closely approaches 100%, the benefit of synergistic caching is marginal. This is particularly illustrated by *mf* and *trace* where, at hit rates upwards of 98%, synergistic caching offers little benefit.

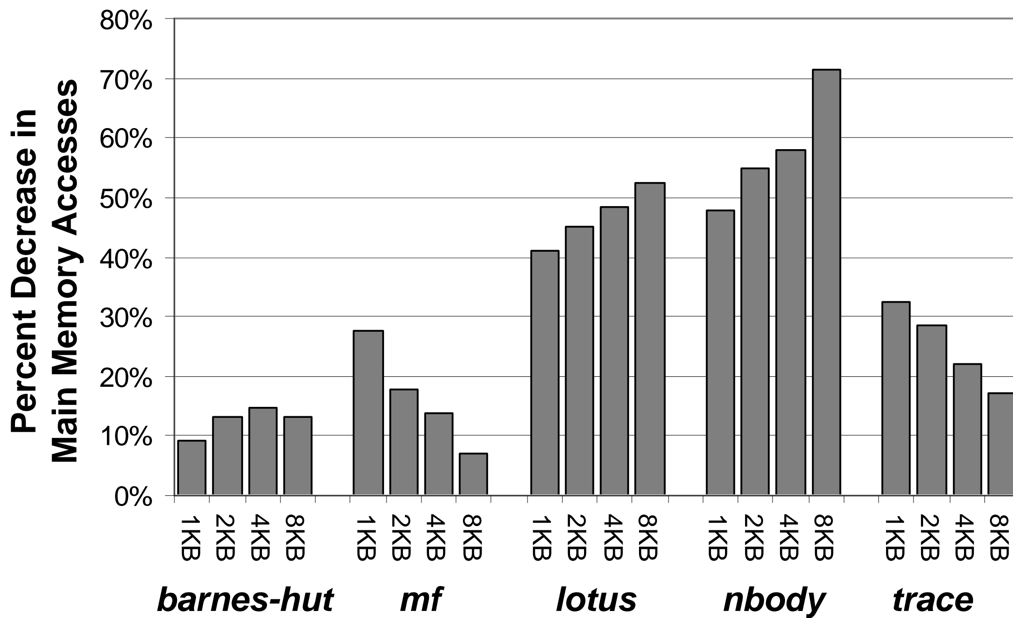

Figure 4.8: Percent Decrease in Main Memory Accesses.

Figure 4.8 shows the percentage of main memory accesses in the traditional L1 system that are serviced by the cluster cache in the synergistic system. For *trace*, which has a small amount of data sharing, as cache sizes increase, the benefit of synergistic caching decreases. This is also true for *mf* which has a small data footprint.

As these applications–with only small amounts of data sharing and a small data footprint, respectively–become less capacity-limited, most memory accesses are retrieved from the local cache, and the benefit of synergistic caching decreases. For these two applications, the total number of main memory accesses at cache sizes above 2KB is only a few percent of the total number of memory accesses.

The second trend is demonstrated by *nbody*. As with the two applications discussed in the paragraph above, *nbody* has a very low miss rate (less than 1%) as cache size increases. However, *nbody* has a larger data footprint than *mf* and much higher percentage of shared accesses (over 90%) than *trace*. As cache capacity increases and misses become only compulsory misses, the miss rates more clearly reflect that the misses in the L1 cache are more likely to be to shared data than to private data.

The third trend seen in the percentage memory access decrease data is that *lotus*, which has a large amount of sharing (see Figure 4.3 from the previous section) and relatively high miss rates (see Figure 4.7, reaps the most benefit from synergistic caching.

The effect of sharing is most notable in *barnes-hut* and *lotus*. In these applications with large amounts of shared data and relatively low hit rates, the number of hits in the cluster cache remains almost constant across cache sizes. On the other hand, applications with little sharing and low miss rates benefit little when the applications are not capacity-limited. This is most apparent in *mf* and *trace* for L1 cache sizes greater than 4KB. At these larger cache sizes, the applications are no longer capacity-limited and, since there is little shared data, they reap little benefit from synergistic caching.

We've shown that the shared data available in applications is in fact retrievable from neighboring nodes. And, a perhaps surprising observation, even applications with little sharing can be benefited from synergistic caching when they are capacity-limited. Synergistic caching effectively increases the capacity of the local cache by

allowing neighboring caches to act as a resource for fast access to recently used data. The benefit is symmetric in that neighboring caches likewise benefit from the effective increase in cache capacity offered by the other caches in its cluster.

4.3.2 Average Memory Access Time

We now analyze the *average memory access times* across each of the cache sizes. The average memory access time is the average time it takes for a processor to retrieve data. Figure 4.9 shows average memory access times for the synergistic and L1 caching systems for L1 cache sizes of 1 to 8KB.

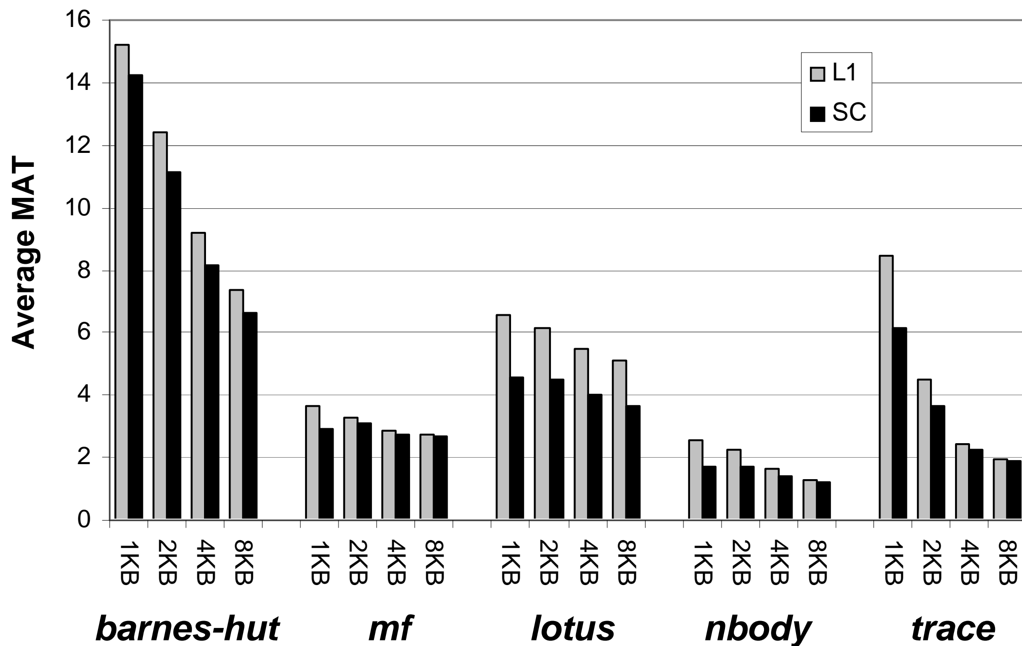

Figure 4.9: Average Memory Access Time.

Figure 4.9 shows average memory access times for the synergistic and L1 caching systems for L1 cache sizes of 1 to 8KB. The trends that showed up in the hit rate discussion given in the previous section also show up in average memory access time.

We now examine the average time to access each level of the memory system—main memory, the L1 cache, and the cluster cache. The average main memory access time is 48 to 50 clock cycles across all applications and cache sizes. It can be less than 50 clock cycles since requests can piggy-back on previously-issued requests to the same data line. The average access time to retrieve data from the L1 cache depends on cache latency rounded up to the next whole clock cycle, as shown in Table 3.4 in Chapter 3.

Figure 4.10 (on the next page) shows the average access time for the cluster cache. Because of contention, the cluster cache access time (CC-AT) may be larger than the ideal network and cache access delay. Without port contention, the range of cluster cache access times are as shown in Equation 4.1. In the equation, $L1_AT$ is the access time of the L1 cache, MD is the manhattan distance between nodes in the cluster, and "s.p." and "l.p." refer to the shortest and longest path, respectively.

$$L1_AT + 2 \times MD(s.p.) <= CC_AT <= L1_AT + 2 \times MD(l.p.) \tag{4.1}$$

The fastest a cluster request will be serviced is by the closest node. The latency for a round-trip access to this node is two times the manhattan distance, the second term in each inequality. At the servicing node, the delay to retrieve the data from the L1 cache must also be taken into account, thus, the first term in each inequality. For example, with a cluster size of 4, the shortest path is a single hop in the network, or 1 processor clock cycle. And the longest path is to the diagonal neighbor, 2 hops or clock cycles.

However, port contention adds additional latency to the access. Figure 4.11 shows details of the time spent accessing the cluster cache. The figure shows cluster cache access time broken down into two categories: (1) the time spent sending, accessing and retrieving the data, and (2) the time spent waiting for the neighboring processor's

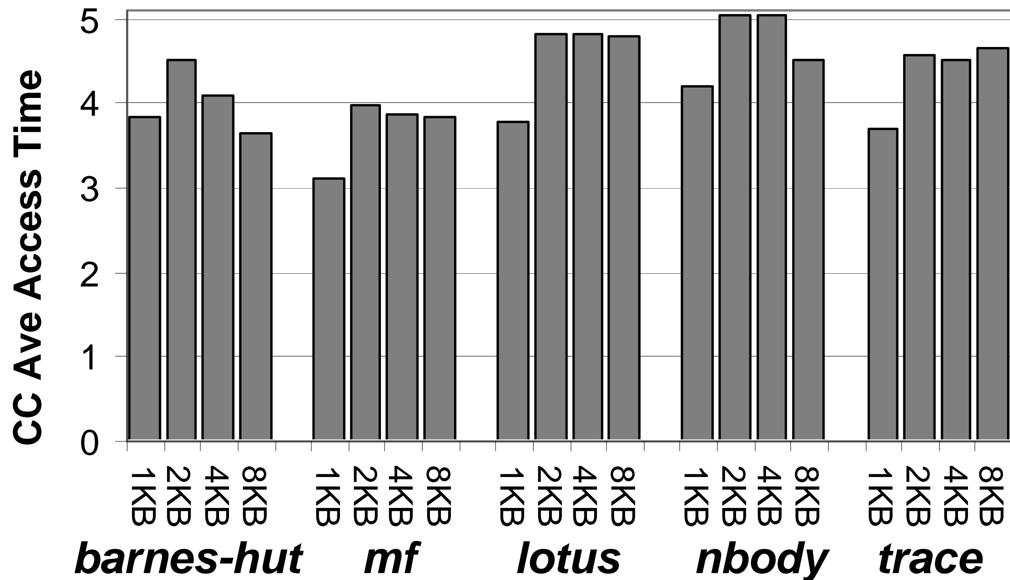

Figure 4.10: Cluster Cache Average Access Time (CC-AT).

port to become free. The data shown in the graph are only for the neighboring cache that actually services the data request. The requests to other nodes in the cluster also add load on their respective cache ports, but the stalling time doesn't add to the access time for the given request. That contention shows up in later requests that are serviced by those nodes. Figure 4.11 shows that only a small percentage of access time is spent stalling at the ports of neighboring caches, which justifies our initial assumptions made in Chapter 3.

In this section we examined the hit rates and average memory access times for the synergistic caching system. We also presented the improvement in these metrics of the synergistic system over the independent-L1 caching system. We now look at how much improvements in these metrics translate into improvement in overall execution time for each application.

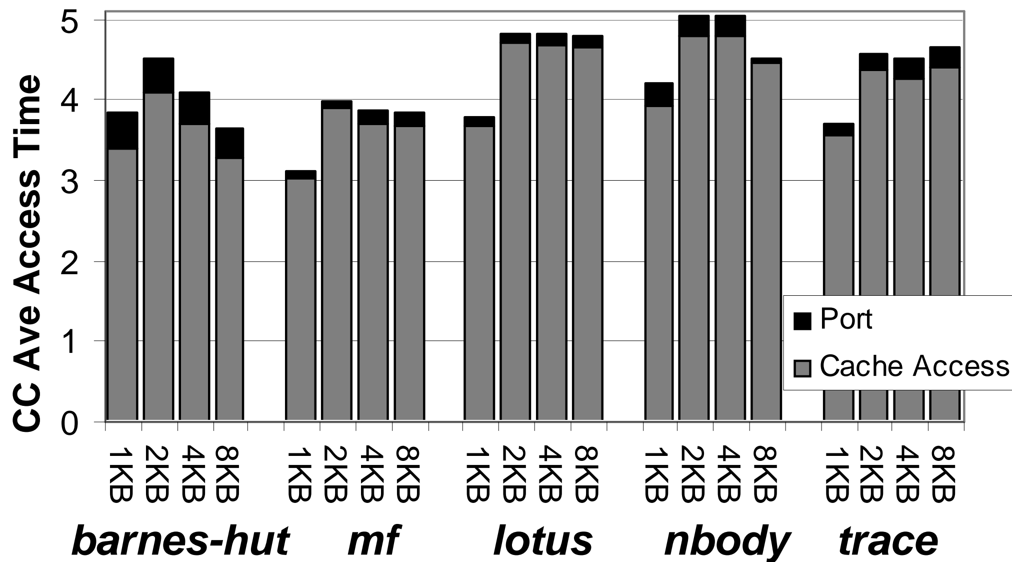

Figure 4.11: Cluster Cache Average Access Time (CC-AT) Details.

4.3.3 Execution Time

In this section, we look at how improvements in hit rates and average memory access times correspond to improvements in overall execution time. Figure 4.12 shows normalized speedup of the L1 caching system and the synergistic caching system. The speedup is normalized to the execution time of the L1 caching system with a 1 KB cache.

To emphasize the difference between the systems, we show the execution speedup of the synergistic system relative to a traditional caching system in Figure 4.13.

We notice three main trends. First of all, the synergistic caching system does well when the applications are capacity-limited. This is the same trend we saw in the hit rates and the average memory access times. This is demonstrated clearly by the increased speedup for smaller cache sizes. For cache sizes of 1 and 2KB, we see speedups of 8 to 28% across all applications. Even applications with little

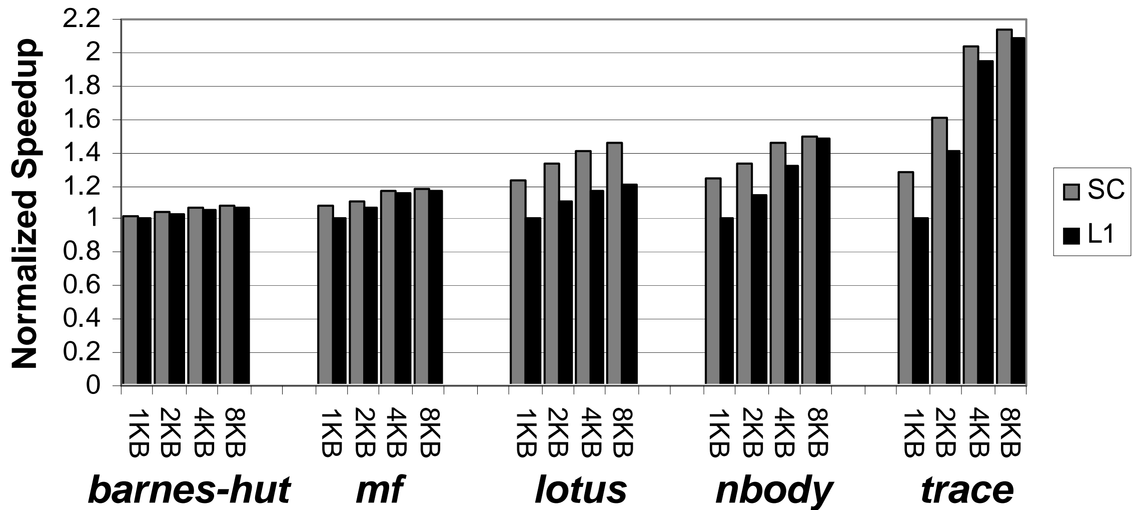

Figure 4.12: Synergistic Caching and Traditional Caching Execution Times.

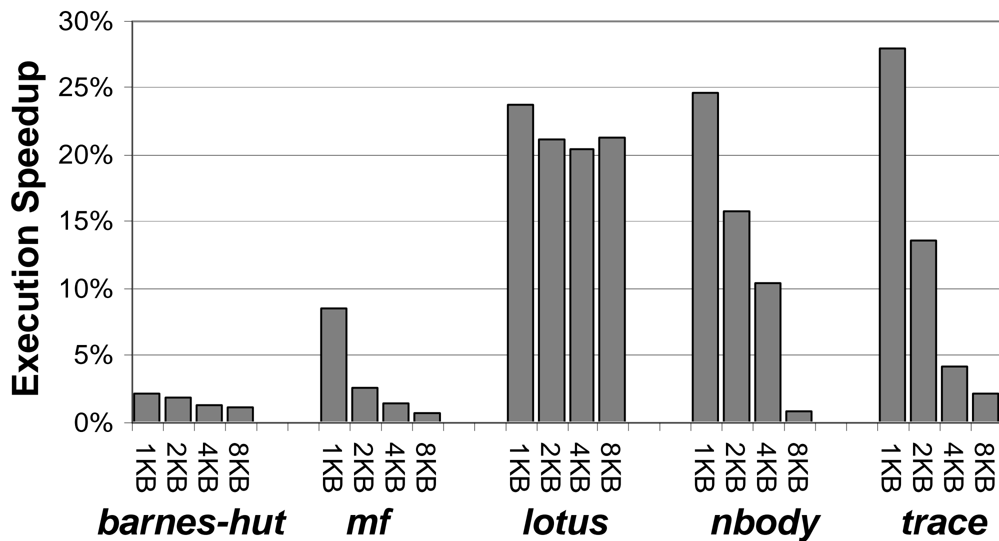

Figure 4.13: Execution Speedup of the Synergistic Caching System over the Traditional Caching System.

sharing, specifically *mf* and *trace*, show performance improvements in these capacity-limited situations. However, as these applications with little sharing no longer become

capacity-limited—with cache sizes upwards of 4KB—synergistic caching is of little benefit.

The second trend we observe is that *lotus*—which has relatively large amounts of shared accesses, 87%—shows performance improvements for all cache sizes. We see over 20% speedup across the range of cache sizes. Again, we noticed this trend in the hit rates as well.

The final trend, however, contradicts observations seen in the hit rate and average memory access time data. *Barnes-hut*, even though it has large amounts of sharing—99% of all accesses are to shared data—we observe little improvement in execution time when the application runs on a synergistic caching system. This is because *barnes-hut* exhibits a high latency tolerance and because many of the data accesses are to non-concurrently shared data. Since *barnes-hut* has multiple threads running on a single node, while one thread is waiting for data, the other threads can be doing useful work[4]. The memory system improves, as evidenced by the increases in hit rates and decreases in average memory access times in Figure 4.7 and Figure 4.9. But this improvement does not translate into decreased execution time since *barnes-hut* is relatively immune to the performance of the memory system.

The question then arises, why aren't all applications written with high latency tolerance? There are two main reasons that applications may not be written with high latency tolerance. The first is, some applications inherently have a small load-use distance. Data-dependent instructions are inherent to the functionality of the application.

The second reason is that context switching between threads can be costly either in time or hardware resources. In a multi-threaded processor, duplicate register sets are used for fast context switching. In a non-multi-threaded processor, current registers

[4]We remind the reader, as stated earlier, that the cost of context switching may also be costly in terms of area or delay. In the simulations, this cost is not modeled.

must be saved to memory and the next thread's registers must be restored. This can take many processor clock cycles. Thus, context switching between threads is expensive in performance time or chip area. Taking this into account, the costs of context switching may outweigh its benefits.

We see the trends examined in the previous paragraphs more specifically in Figure 4.14 where we show detailed execution cycles—the worked cycles, when the processor executes an instruction, and the stalled cycles—idle cycles when the processor is waiting for data. The performance of the L1 system is shown in the left bar for each cache size, and the synergistic system is shown in the right bar.

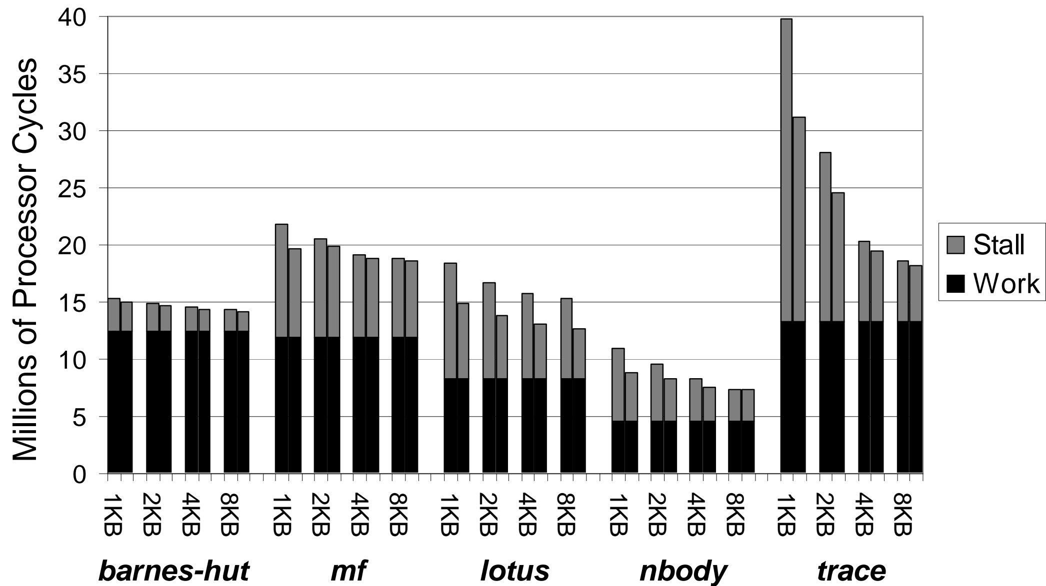

Figure 4.14: Synergistic Caching System Detailed Execution Time.

The percentage of stall cycles decreases dramatically between configurations for capacity-limited applications and is consistently less for the synergistic system running *lotus*, an application with large amounts of shared data. However, as before, *barnes-hut* shows little improvement in the percentage of stall cycles.

Trace and *barnes-hut* partition data to minimize sharing dependencies between

threads. In *trace* the partitioning is done spatially, with little sharing between threads. In *barnes-hut* the partitioning is done temporally, with most of the data shared non-concurrently. This partitioning construct makes sense in an all-or-nothing memory system where non-local data take a long time to retrieve. However, with the graded-delay memory system and high-bandwidth offered by the CMP system, these restrictions can be relaxed. Applications with large amounts of shared data and thread interaction can be accommodated and, in fact, encouraged.

4.3.4 Summary

In this section we examined the synergistic caching system and compared it to a traditional caching system. We showed that it held to the predictions given in Chapter 3. Specifically, we showed that the system performs particularly well in the following three cases: when there are large amounts of across-chip sharing in the application, when the application is capacity-limited, and when the application shows low latency tolerance.

We also showed that even though the underlying memory system improves, this may not directly translate into improved execution time because of the non-blocking memory system and high latency tolerance in applications such as *barnes-hut*. Further analysis of the costs of context switching – area and delay – should be explored further.

In the next section we introduce variations in duplication modes of the synergistic caching system and analyze each of their performance.

4.4 Duplication Modes

The synergistic caching system benefits from varying duplication modes in a cluster cache. In this section, we examine the effects of each duplication mode—*beg*, *borrow*, and *steal*—on the performance of the synergistic caching system.

As mentioned in Chapter 3, each duplication mode offers a capacity-latency trade-off. The *beg* option minimizes the access time to repeatedly access shared data, but it also has the minimal worst-case cluster cache capacity. The *borrow* and *steal* options, on the other hand, maximize cluster cache capacity, but, because there is no duplication in the cluster cache, shared data that are repeatedly accessed in a neighboring node's cache incur incrementally large penalties.

In the following sections we explore the effects of each duplication mode on hit rate, average memory access time, and execution time. We show that, depending on the application and the data footprint of the application, the optimal duplication mode changes.

4.4.1 Hit Rates

Figure 4.15 shows the hit rates of each application across cache sizes of 1KB to 8 KB for each duplication mode. The figure shows which level of the memory system services each access—the L1 cache, the cluster-cache in the synergistic caching system, or main memory. As before, these levels are denoted as L1, SC, and MM, respectively, in the graph.

Three main trends are apparent. First, the *overall* hit rate—the percent of accesses serviced by any system other than main memory—is similar across all duplication modes for most applications and cache sizes.

The notable exception is the overall hit rate for the *borrow* duplication mode for small cache sizes, which exceeds that of the other modes. This second trend is evidence of the increased effective cache capacity offered by the beg mode.

The third trend is the difference in the systems servicing requests. With *borrow* and *steal* duplication modes, many more requests are serviced by the cluster cache instead of the L1 cache or main memory. These modes also show increases in the total number of accesses serviced by the caching system—particularly noted in *lotus*, *nbody*,

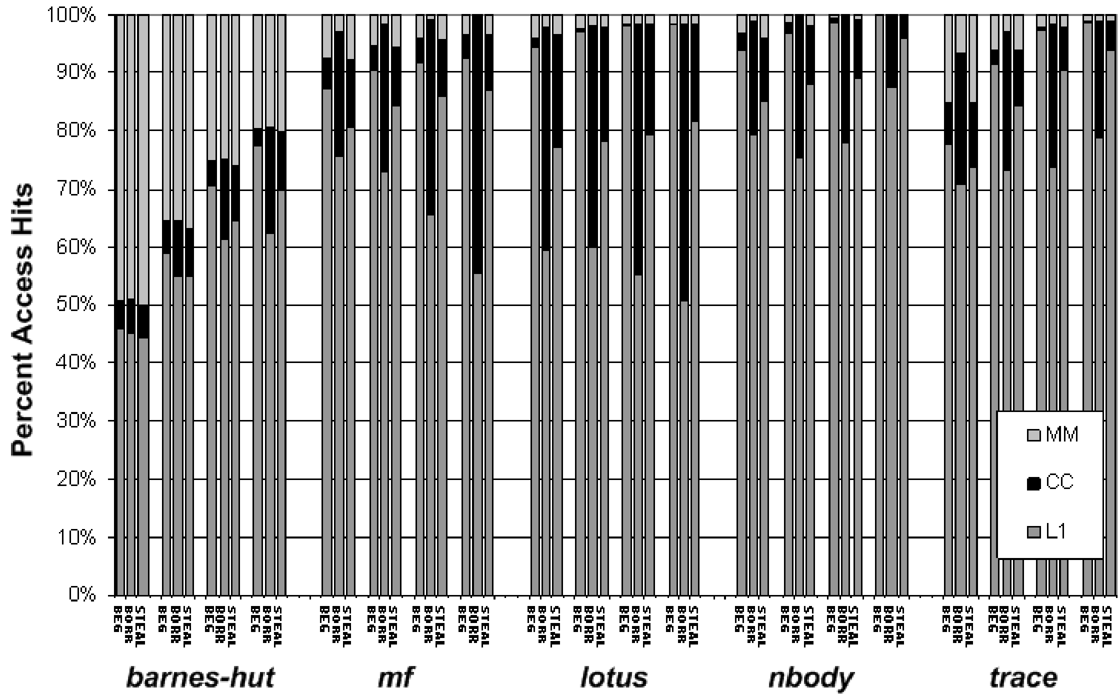

Figure 4.15: Hit Rates of the Duplication Modes.

and *trace* running with small cache sizes. But the penalty for retrieving data from the cluster cache may not be balanced by the benefit of increased effective capacity. When this is the case, the performance of the application decreases.

4.4.2 Average Memory Access Time

In this section, we discuss the effect each duplication mode has on the average memory access time of the system. Figure 4.16 shows the average memory access times for each duplication mode.

To emphasize the difference between the modes, we show Figure 4.17 which is the average memory access time speedup normalized to the L1 system. With capacity-limited applications, the effective increase in capacity of the *borrow* mode improves the average memory access time. This is most notably shown in *lotus*, *nbody* and

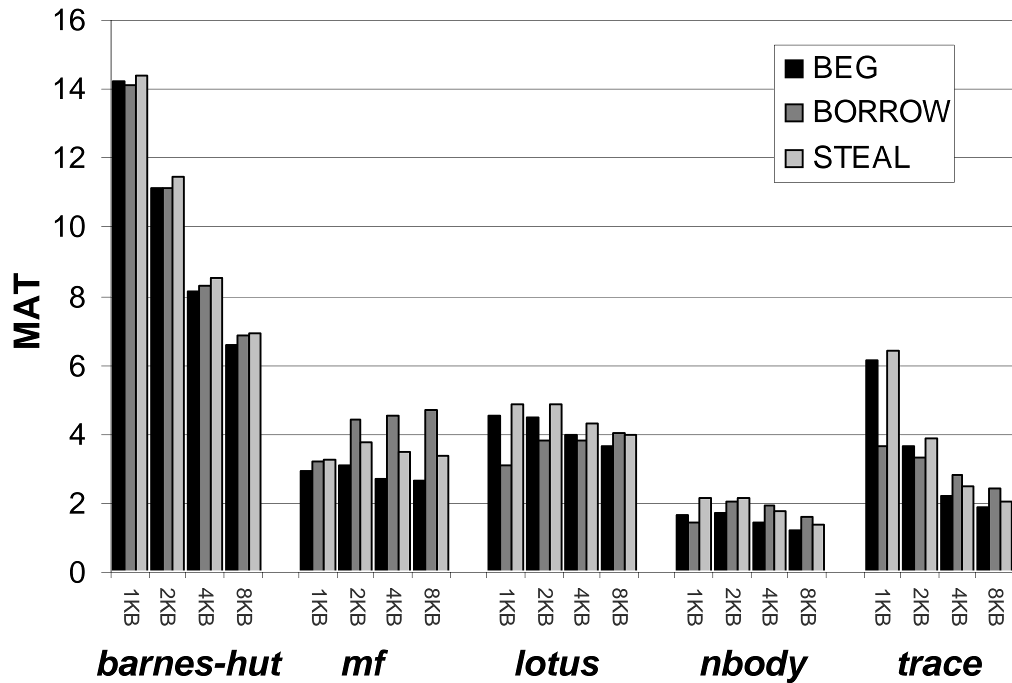

Figure 4.16: Average MAT across duplication modes.

trace for cache sizes of 1KB.

Cluster Cache Average Access Time (CC-AT)

The average time to access the cluster-cache varies as demand on cluster-cache ports increases. Figure 4.18 shows the average access time to retrieve data from the cluster cache.

For applications that aren't capacity-limited—like *mf*—the non-duplicating modes, *borrow* and *steal*, have larger access times. This effect is also seen in the 4KB and 8KB systems running *barnes-hut* and *nbody*. As seen in the hit rate data in Figure 4.15, using the *borrow* and *steal* modes, accesses that were serviced by the L1 cache in the *beg* mode are now serviced by the cluster cache. This increased demand on the cluster cache increases the average cluster cache access time due to port contention.

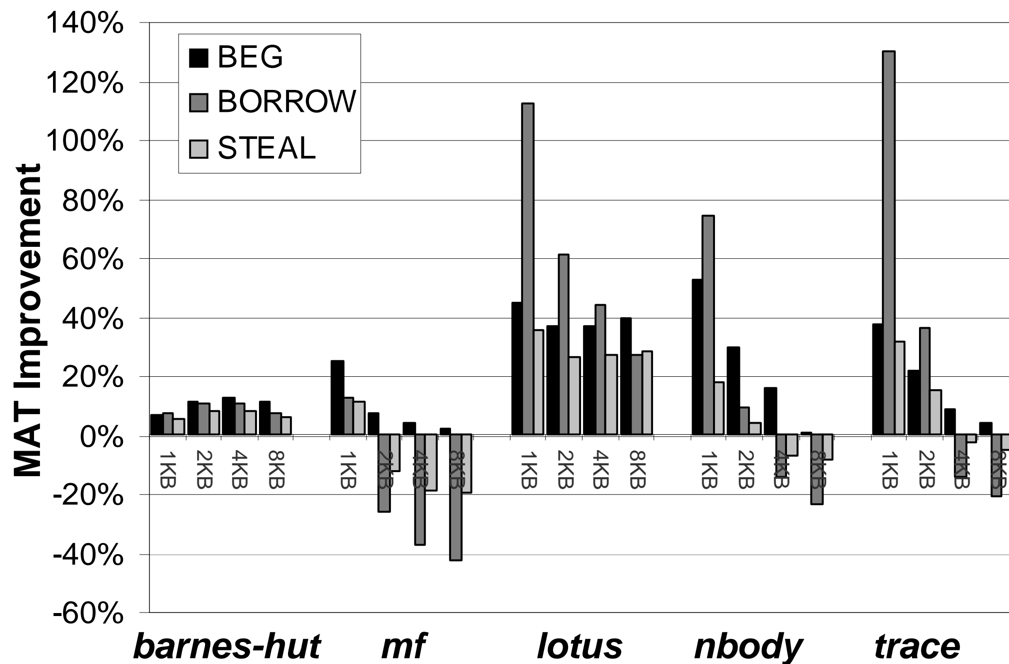

Figure 4.17: Normalized average MAT for each duplication mode.

Figure 4.19 shows the average number of cycles per cluster cache access spent waiting for a port at the cluster cache. Across all applications and duplication modes, up to 45% of the time the request stalls at the neighboring cache port. Because of the increased frequency of cluster requests in the *borrow* and *steal* modes, the port contention can more than double that of the *beg* mode. This effect is most clearly seen in *mf* which, because if its small data footprint, is not capacity-limited.

Contention at the port of the neighboring cache occurs for two main reasons. Either the owning process is using the port or other cluster processors are vying for access to the cache. Figure 4.20 shows the percent memory accesses per instruction for each application.

The owning processors issue memory requests on average only 19% to 34% of the time. And, with memory stall cycles taken into account, the percentage of run

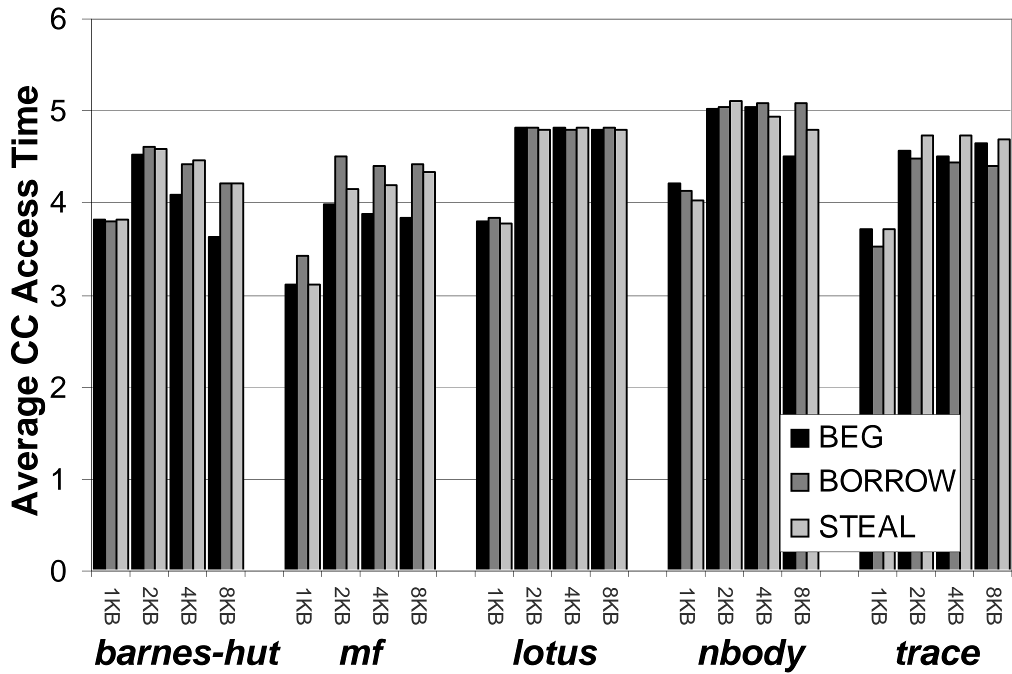

Figure 4.18: Cluster-cache Average Access Time (CC-AT) Across Duplication Modes.

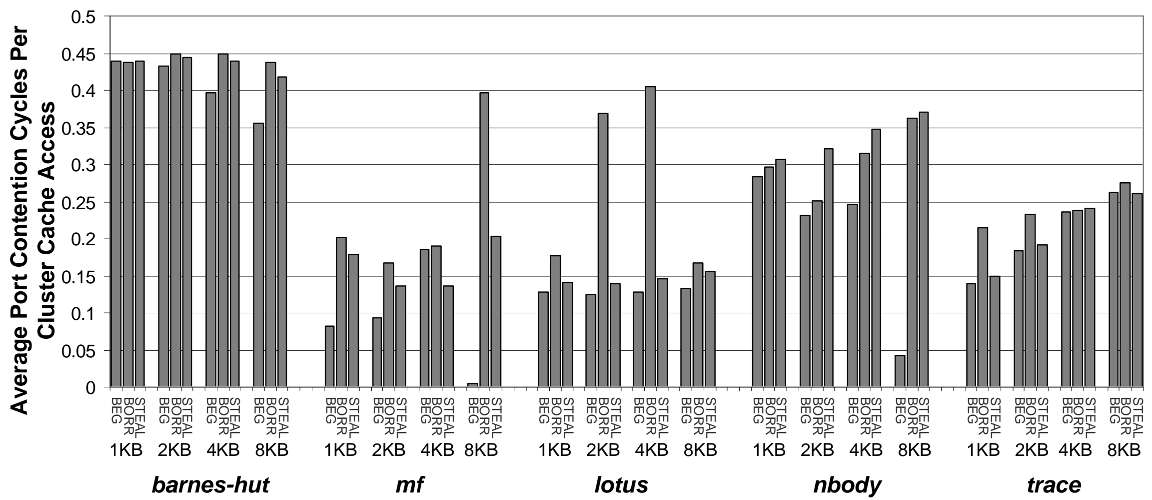

Figure 4.19: Port Contention Cycles at the Cluster Cache.

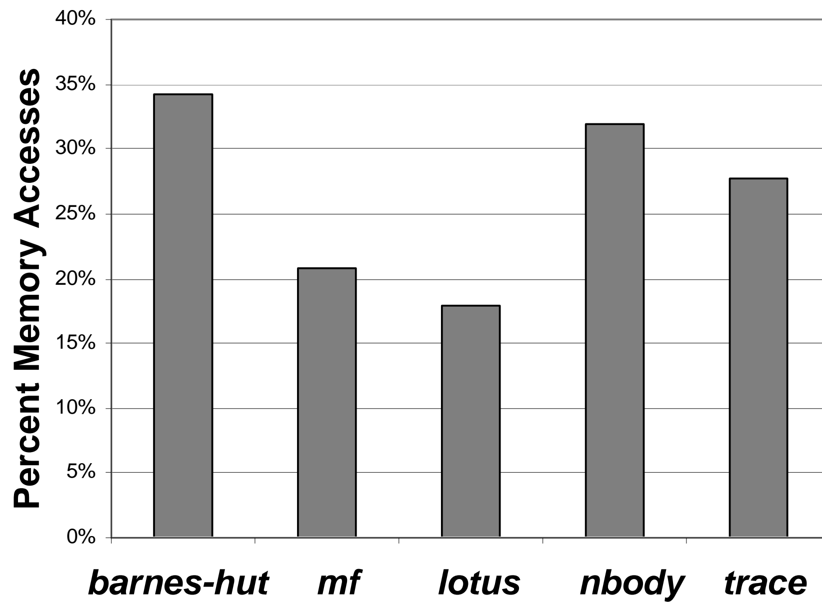

Figure 4.20: Instruction Mix.

cycles that the processor is requesting data is much smaller. So, this implies that neighboring processor requests are making up the difference to keep the ports busy up to 50% of the time.

These figures show that average memory access time doesn't always correlate with the overall hit rates of each duplication mode. As capacity becomes abundant, the *borrow* and *steal* modes can perform even worse than traditional caching systems despite higher overall hit rates. The added effective capacity of these systems fails to counterbalance the increased access time of frequently used data held by neighboring caches. The increase in the number of cluster accesses of the *borrow* and *steal* modes also causes increased port contention in non-capacity-limited scenarios.

In the next section, we look at how the hit rates and memory access times of each duplication mode translate into execution speedup.

4.4.3 Execution Time

Figure 4.21 shows the effects of each duplication strategy on overall execution time. Again, we show execution speedup of the synergistic caching system relative to a traditional caching system for each duplication option.

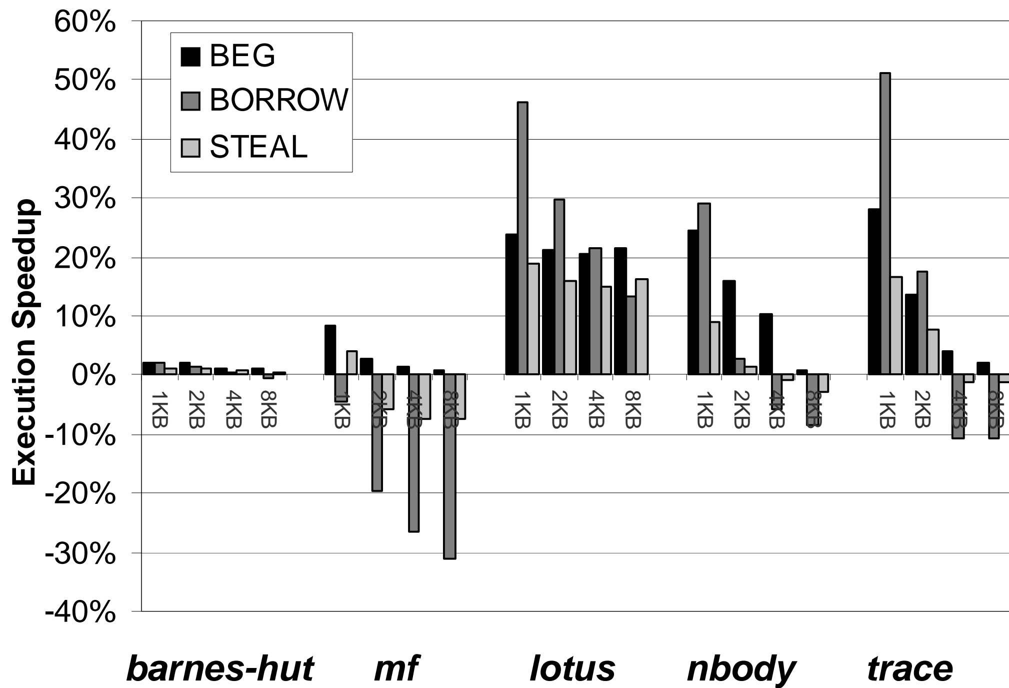

Figure 4.21: Execution Speedup for Duplication Modes as Compared to an Independent-L1 Caching System.

From the graphs, we notice six pronounced trends. First, as we saw in the baseline system, *barnes-hut* is affected very little by changes in the memory system because of its high latency tolerance.

Second, the *borrow* mode performs best when the application is capacity-limited. This is shown in the configurations with smaller cache sizes, particularly for *lotus*, *nbody*, and *trace*.

However, borrowing is not advantageous when an application is not capacity-limited. This third trend is most noticeable in the *mf* application. *Mf* does not benefit from the extra capacity. Instead it is penalized for not duplicating shared data in the requesting cache. In the case of *mf*, the *borrow* duplication mode performs worse than both of the other duplication modes for all cache sizes. The *beg* duplication mode never retrieves shared data that is held by a neighboring cache into the local cache.

In the fourth trend, we see that the *steal* mode retrieves the data into the local cache while still maintaining a single copy of data in the cluster cache as is done with the *beg* mode. Subsequent repeated requests to that data can then be serviced by the local cache as long as it is not "stolen" by a neighboring cache. In this mode, the caching strategy of taking advantage of temporal as well as spatial locality is maximized while still maximizing overall cluster cache capacity. This key difference between *borrow* and *steal* modes is most pronounced in the larger cache sizes for *mf*, *nbody*, and *trace*.

The fifth trend is that, while the *steal* mode performs better than the *borrow* mode in non-capacity-limited scenarios, it is never the best-performing of the three duplication modes. In capacity-limited situations, it is out-performed by both the *beg* and *borrow* modes. And in non-capacity-limited situations, it is out-performed by only the *beg* mode. The *steal* mode, however, often performs better than the independent L1 caching system because of its effective increased cluster cache capacity. However, as capacity becomes less of an issue, the *steal* mode performs poorly, up to 8% worse than the L1 caching system as shown by the *mf* application at an L1 cache size of 8KB.

The *steal* mode is beneficial when data sharing is non-concurrent. In that case, a given node finishes using the data before a thread on a neighboring node begins to consume the data. So stealing the data from the neighboring node would be beneficial

for two reasons. It would allow for quick access by the requesting node as well as free up space in the supplying node's cache. However, the only application that showed significant amounts of non-concurrent sharing was *barnes-hut* which we've already shown is relatively immune to changes in the memory system.

Finally, the last trend is that the beg mode performs well across all cache sizes and applications. This mode balances the added capacity of the cluster cache with quick access to frequently used data.

While *beg* has a poor worst-case capacity, this worst-case scenario only occurs if all of the memory accesses are shared data and if the same shared data are being accessed concurrently. The performance of the *beg* duplication mode shows that the worst-case capacity does not occur in the set of applications tested. If the worst-case capacity had occurred, the applications would have performed the same as in a traditional caching system.

However, due to duplication of data in neighboring nodes, there is a small amount of redundant data in the cluster cache. However, the sub-optimal use of the total capacity in the cache is outweighed by the fast access to frequently used data.

Switching from the *beg* mode to the *borrow* mode, depending on how capacity-limited the application is, would maximize the benefit of synergistic caching. As mentioned in Chapter 3, this could be implemented by a counter on the number of hits in the level-1 cache. Upon detecting a low hit rate, the system could then easily change to the *borrow* mode. If the hit rate does not improve, or gets worse, after a set number of cycles, the mode could be reverted back to the *beg* mode. If there is little difference between the modes, the system can easily oscillate between the modes since there is no difference in the functionality of retrieving data. If oscillation between modes is not desired, a control register could detect a number of cycles in a given period, and set a default mode.

Similarly, if the user or the compiler has incorrectly specified to use the *borrow*

mode for duplication, upon detecting a low hit rate in the level-1 cache, the system could switch to the *beg* mode. If, after switching to the *beg* mode, the hit rate gets worse, the system can then revert back to the *borrow* mode.

Detecting the hit rate and feeding it back to the cache controller can be performed continuously throughout the execution of the application. In this way, portions of the code that are capacity-limited can perform in the *borrow* mode and the rest of the code can reap the advantage of the *beg* mode.

In the previous sections we showed that the *beg* duplication mode performs best across all applications. This mode balances the added capacity of synergistic caching with quick access to frequently used data. The *borrow* mode performs best in capacity-limited applications but is penalized for frequent L1 misses in non-capacity-limited situations. In a non-capacity-limited scenario, the small penalty for L1 misses overwhelms the benefit of extended capacity.

In the set of applications we explored, *steal*ing proved to be a generally poor-performing option. In some cases it performed better than the independent-L1 system and the *borrow* mode, but it never out-performed the *beg* duplication mode. The *steal* duplication mode proved to be parasitic instead of synergistic–except for the situations noted in the above text. The *steal* mode essentially evicts data that the neighboring node may still be using.

Being able to switch from the *beg* mode to the *borrow* mode, depending on the capacity-limited status of the application, would maximize the benefit of synergistic caching.

4.5 Summary

In this chapter we introduced the experimental set-up we used to simulate the synergistic caching system. We also introduced a set of fine-grain multi-threaded applications to test the system. We first examined each application and looked at the characteristics that define them, namely: the percentage of memory stall cycles, the overall amount of shared data, and the amount of shared data available on neighboring nodes.

We then analyzed the performance of each of the applications running on the synergistic caching platform. These were simulated using the system described in the experimental set-up section. We found that the synergistic caching system performs best in the cases we proposed in Chapter 3. Specifically, the synergistic caching system is beneficial when there are large amounts of sharing in the application, when the application is capacity-limited, and when the application shows low latency tolerance.

We looked at how the three duplication modes—*beg*, *borrow*, and *steal*—affect the overall performance of the system. We examined the hit rates, memory access times, and overall execution times of the system using the different duplication modes. We found that the *beg* and *borrow* duplication modes were the most viable options across all applications and cache sizes. The *borrow* mode was beneficial in capacity-limited situations, and the *beg* mode was beneficial in non-capacity-limited scenarios. The *steal* duplication mode, on the other hand, never performed better than both of the modes. We proposed that adapting between the two modes of *beg* and *borrow* would allow the hardware to adapt to changing application characteristics.

Chapter 5

Shared-L2 Comparison

In this chapter we compare the synergistic caching system to the shared level-2 caching system, another strategy that has been introduced to take advantage of sharing in multiprocessors. In this system, a secondary cache, that is larger and slower than the L1 cache, is shared by all processors in a cluster.

The shared L2 system has a hierarchical structure with two levels of caching. Upon a request for data, a processor checks the first-level cache. If the data are not held by the first-level cache, the processor then checks the second-level cache. Upon a miss in the second-level cache, the processor retrieves the data from main memory.

Figure 5.1 shows a 2-node cluster of the L2 caching system. Each node has its own L1 cache and shares the L2 cache controller and L2 cache. The request to the L2 cache takes the place of the cluster request in the synergistic system.

As with the synergistic caching system, the shared-L2 system may also suffer from port contention. All of the nodes in the cluster compete for access to the port(s) of the L2 cache. With L2 caching there is also an additional level of redundancy. Now, there are not only copies of the data in main memory and one or more of the L1 caches, but there is also another copy of the data in the L2 cache. Finally, for an inclusive L2 cache, there's a possibility of interference in the cache, with one processor

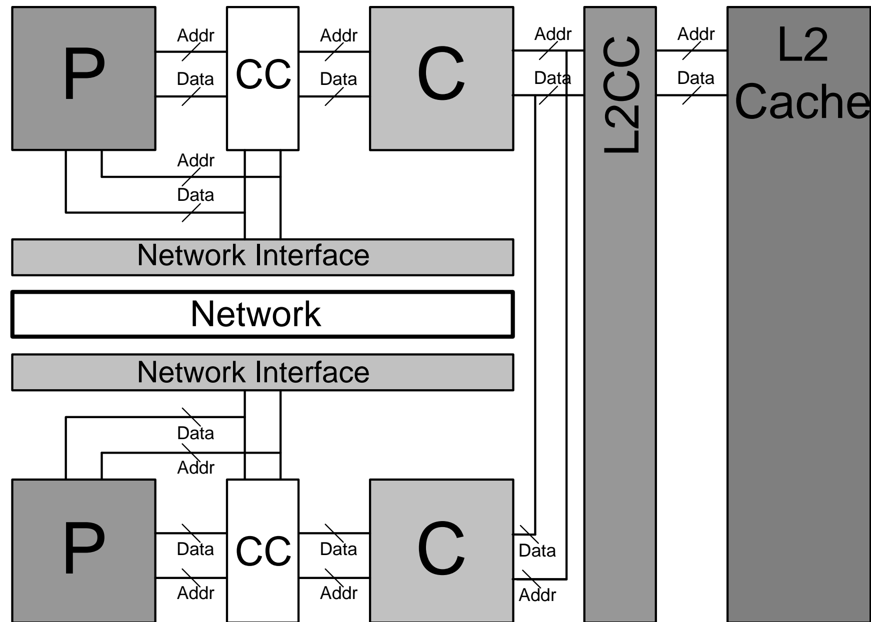

Figure 5.1: A 2-node Cluster of an L2 Caching System.

writing over data needed by a competing processor. In this case, instead of enabling sharing, the L2 hinders sharing as well as data re-use by the victim processor. The possibility of interference also depends on the set-associativity of the cache.

5.1 Area

In comparing the synergistic and shared-L2 caching systems, we use the total area used by each system as the common factor. Figure 5.2 below shows area equivalents for three configurations, C1, C2, and C3.

Area estimates were obtained running Cacti 3.0 in a 650nm technology [26]. In the synergistic system, each L1-cache is 2-way associative and dual-banked. In the L2 system, each L1-cache is 2-way associative and single-banked; each L2 cache is 4-way associative with the number of banks equal to the cluster size[1].

[1]We note that the area calculations found using Cacti are an approximation and may not represent the technology used to build L2 caching systems. Many L2 systems are built to maximize capacity

	SC			L2				
Config	L1 Size	L1 Delay (c.c.)	Total Cluster Area (mm^2)	L1 Size	Delay (c.c.)	L2 Size	L2 Delay (c.c.)	Total Cluster Area (mm^2)
C1	4KB	2	0.50	1KB	1	8KB	2	0.67
C2	8KB	2	0.72	2KB	1	16KB	2	0.82
C3	16KB	2	1.1	4KB	2	32KB	3	1.0

Figure 5.2: Area estimates.

The total cluster area given in the table is the area required for the caching system of a single cluster, with a cluster size of four. This includes the area of all of the L1 caches and, with the shared L2 system, the L2 cache. In the L2 system, L1 sizes are limited since area is sacrificed for L2 cache area.

5.2 Performance

In comparing the performance of the synergistic and shared-L2 caching systems, we explore how effective each system is at allocating and retrieving the shared data shown in Figure 4.3 of Chapter 4.

We show performance results for the synergistic caching system with L1 cache sizes ranging from 4KB to 16KB. The L1 cache sizes for the L2 system range from 1KB to 4KB, and the respective L2 cache sizes range from 8KB to 32KB. A comparison of the areas for the synergistic and L2 systems across cache sizes is found in Figure 5.2 in the previous section. We use this range of cache sizes to highlight the difference in performance for each application's data footprint given in Figure 4.1 in Chapter 4.

and not delay. Thus, compared to the L2 area and delay figures shown in Figure 5.2, actual L2 caches may be more dense but also slower. The numbers shown in Figure 5.2 were used during simulation as an approximation.

In the following sections, we use the *beg* duplication mode for the synergistic caching system, and the L2 cache is an inclusive cache where all data held in the caches of cluster nodes are also held in the L2 cache.

5.2.1 Hit Rate

We now look at the hit rates of the shared-L2 system and compare them to those of the synergistic caching system. Figure 5.3 shows the hit rates of each system across the three configurations described in Table 5.2.

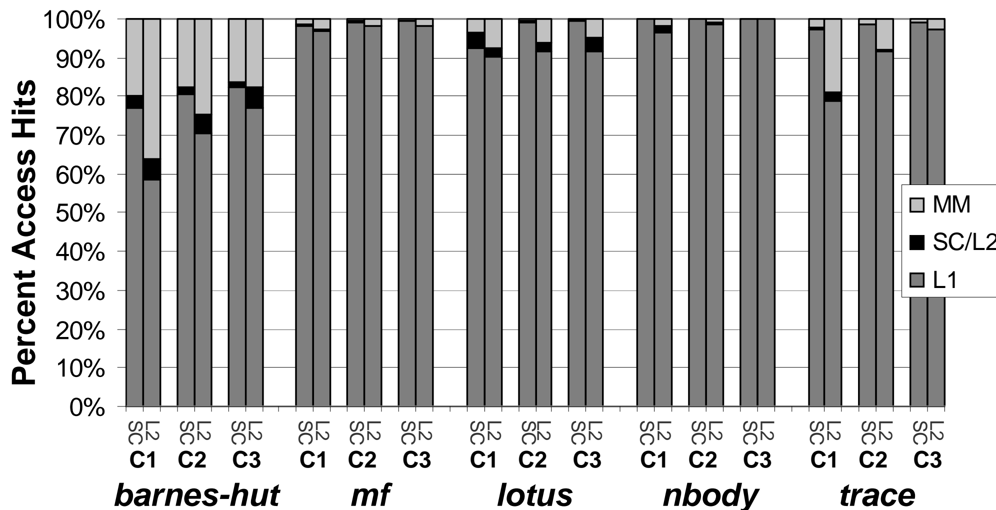

Figure 5.3: Hit Rates for the Synergistic and L2 Caching Systems.

The major trend shown in Figure 5.3 is that as cache sizes increase, the difference between the L1 system and the shared-L2 system decrease. However, for smaller cache sizes, C1 and C2, the difference between the two systems can be dramatic. The graph shows that the synergistic caching system has up to a 16% difference in overall hit rate for the smaller configurations.

To understand these effects, we look at the L1 hit rates for a traditional L1 caching system. Figure 5.4 shows the L1 hit rates for all applications across L1 cache sizes of

1KB to 32KB.

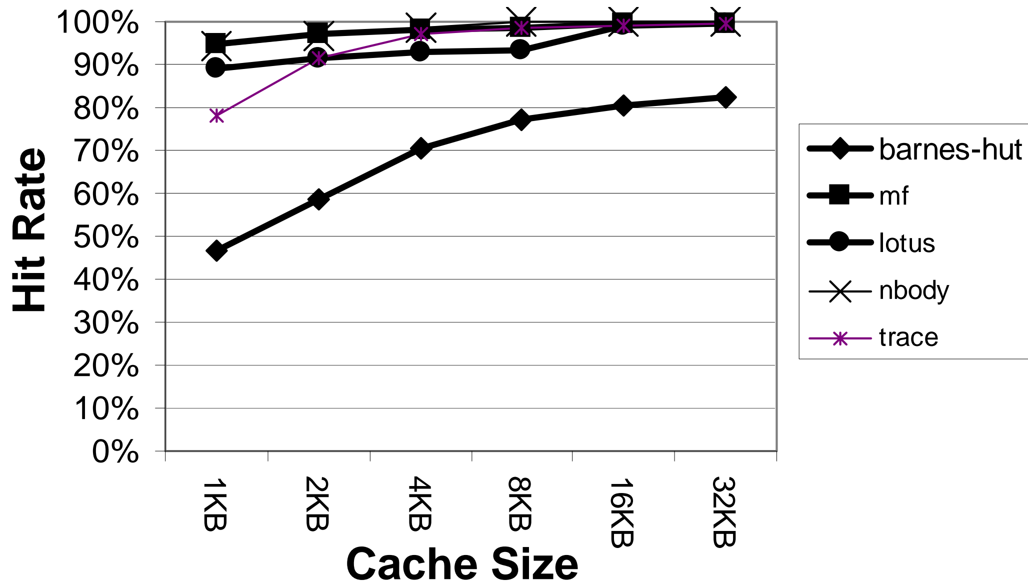

Figure 5.4: Hit rate for the L1 caching system.

With the C1 configuration, when the synergistic caching system is operating at the 4KB point on the L1 hit rate curve, an equivalent area shared-L2 system is operating at the 1KB point, where the hit rate value is not only much lower, but the curve itself is much steeper. This steepness indicates the application is capacity-limited.

However, as both systems begin to operate further out on the L1 hit rate curves, the hit rate curves becomes flat, indicating that the application is no longer capacity-limited. When this is the case, the L2 system is no longer limited by small L1 caches, where small is in comparison to the size of the working set. The only difference then between the two systems is the time to access the shared data in the cluster-cache and L2 cache, respectively.

Again we look at Figure 5.3 which shows the hit rates of each respective system. For the smallest cache area, we see that the overall hit rate for the L2 system is less than the synergistic caching system. But, more importantly, many of the hits in the

L1 cache in the synergistic caching system are replaced by hits in the L2 cache in the shared-L2 system.

However, as the conglomerate cache area grows larger and both systems begin to operate on the flat portion of the L1 hit rate curve, both systems exhibit the same amount of hits in each level of the hierarchy. The difference between the systems becomes less pronounced.

5.2.2 Average Memory Access Time

Figure 5.5 shows the average memory access time across applications. The synergistic caching system shows lower average memory access times than the L2 system for all configurations.

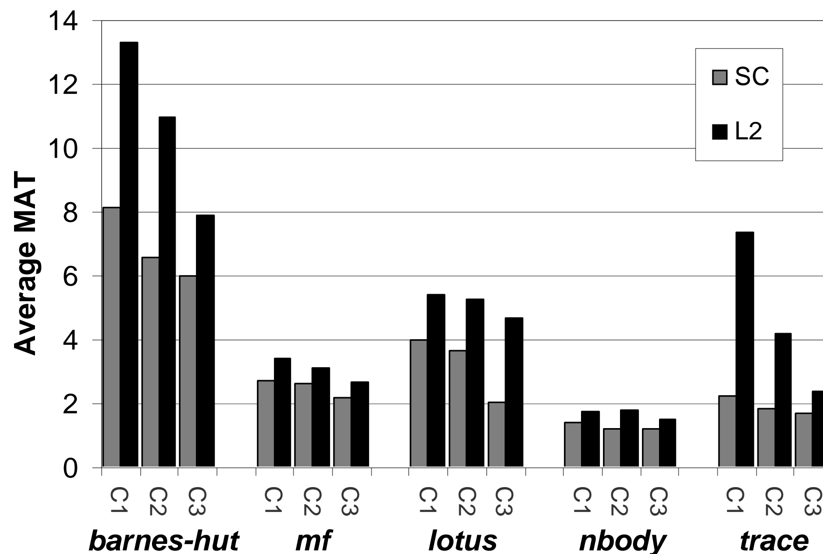

Figure 5.5: MAT Comparison of SC and L2 Systems.

In the C1 configuration, the average memory access times for the synergistic caching system are significantly faster than the L2 system. Particularly in the C1

configuration, the average memory access times for the synergistic system are significantly faster than the L2 system. For example, as shown in Figure 5.5 under the C1 configuration, the *trace* application has an average MAT more than three times faster than the L2 system.

This smaller average memory access time in the synergistic caching system is due to the increased frequency of memory accesses in the L2 system. It is also because many accesses serviced by the L1 cache in the synergistic system are instead serviced by the L2 cache in the L2 system.

The data in Table 5.2 shows that for the two smaller configurations, C1 and C2, the L1 access time for the shared-L2 system is less than that of the synergistic system. However, as shown in the hit rate data of Figure 5.3, the decreased capacity overwhelms the advantage of faster L1 access time. This is evidenced by the large number of accesses that must be serviced by the L2 cache instead of the L1 cache, as shown in Figure 5.3.

Figure 5.6 shows the average access times for the cluster cache and L2 cache in each respective system. The cluster cache access time includes the time to miss in the L1 cache, send the request via the on-chip network, access the neighboring cache, and return the data through the network to the requesting node. Cluster cache access time also includes cycles spent contending for the neighboring cache's port.

The time to access the L2 cache is the wire delay to and from the centrally located L2 cache controller plus the access time of the L2 cache. If there is contention at the L2 cache port, this stall time is also added to L2 cache access time.

Figure 5.6 shows that the effective access time of the cluster cache is 1 to 1.5 cycles greater than the L2 access delay. This added delay is due to traversing the network as well as a small increase in port contention. For configurations C1 and C2, there are also increased L1 access times, as shown earlier in Table 5.2.

Figure 5.6 shows a breakdown of each of the components of L2 and cluster-cache

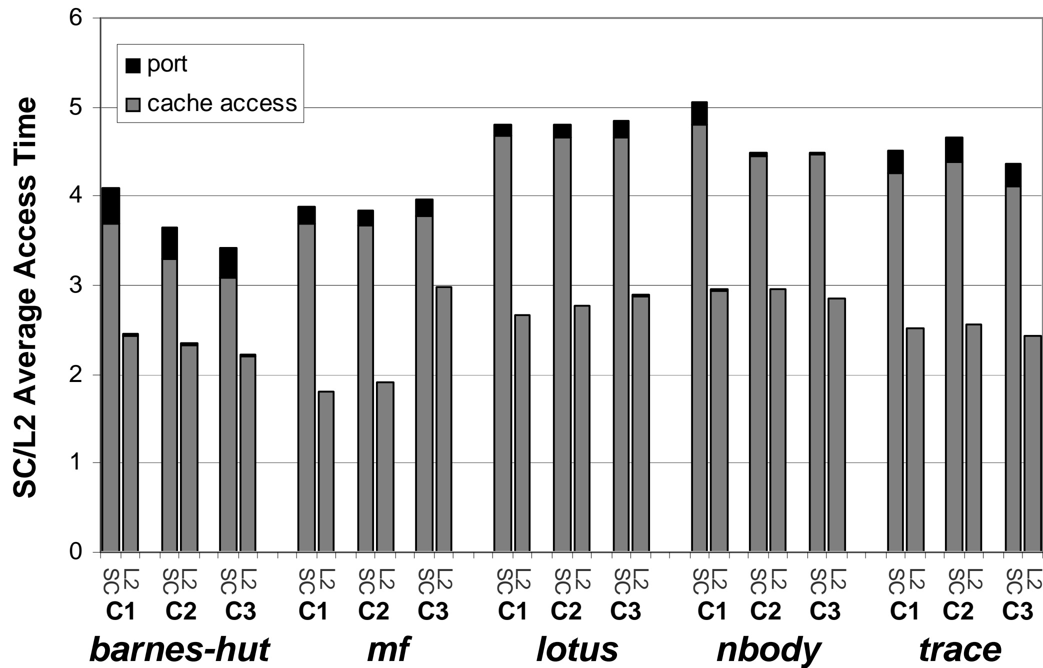

Figure 5.6: Average access times for the cluster and L2 cache.

access time. For each system, the time to get to the cache—the network delay for the cluster-cache, and the wire delay for the shared-L2 cache—is included in the access time of the cache.

Two main characteristics about port contention and cache access time are evident in Figure 5.6. First, port contention decreases in both systems as demand on the secondary systems, the cluster-cache and the shared-L2 cache, decreases.

Second, across all cache sizes and applications, time spent due to port contention is greater for the synergistic system than for the L2 system. For a single L2 access, on average a small fraction of a cycle is spent waiting for the L2 port. For the same tests of the synergistic caching system, on average up to a quarter of a cycle is spent waiting for a neighboring cache's port. This effect is because the cluster requests must compete with requests sent by the local processor for access to the neighboring

cache's port.

However, despite the increase in port contention, this effect shows little net effect on the overall memory access times shown in Figure 5.5. The average memory access time of the L2 system is disadvantaged because of the higher frequency of L2 and main memory accesses. Many L1 hits in the synergistic caching platform are replaced by L2 hits or main memory accesses in the shared-L2 platform. Thus, the overriding limit for the shared-L2 system is the limited L1 cache sizes.

5.2.3 Execution Speedup

We now look at the total execution time for the synergistic and shared-L2 caching systems. Figure 5.7 shows the execution times of the applications when simulated on the respective systems.

The x-axis shows each application across all three configurations, and the y-axis is the execution time in millions of clock cycles. For further insight, we show the speedup of the synergistic caching system over the L2 caching system for each configuration in Figure 5.8.

Three trends are evident in Figures 5.7 and 5.8. The first trend, that we also saw in the comparisons against the independent-L1 system in Chapter 4, is that *barnes-hut* is resistant to changes in the memory system because of its high latency tolerance. Figure 5.8 shows little difference between the performance of *barnes-hut* on either the synergistic or L2 system.

The second trend is that, again, synergistic caching shows the highest speedups when the application is capacity-limited. We saw this same trend when comparing the synergistic system to the independent-L1 system, but it is even more pronounced in comparison with the L2 system because of the small L1 cache size of the L2 system. The synergistic system allocates all of the cache area to L1 caches, while the shared-L2 system must partition the area between the L1 and L2 caches. The

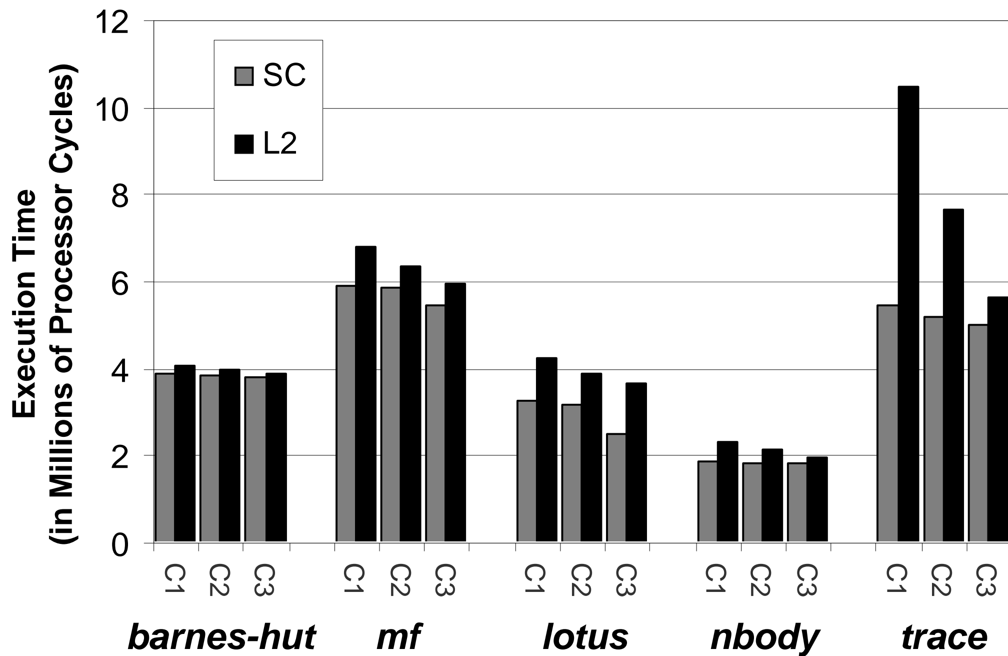

Figure 5.7: Execution time of synergistic and shared-L2 caching systems.

subsequently smaller L1 caches of the L2 system exacerbated the already capacity-limited applications. This is most notable in the C1 configuration of *trace* where the synergistic system out-performs the L2 system by 92%.

As we saw in our analysis of hit rates, shown in Figure 5.4, for cache sizes of 4KB and higher, the synergistic system is operating at the flat area of the curve. The L2 system, however, is operating at a much steeper region of the L1 hit rate curve. Data that are serviced by the L1 cache in the synergistic system are serviced by the L2 cache or, in the worst case, by main memory, in the L2 system. Figure 5.8 shows how this directly effects an increased execution time.

As cache sizes increase and the applications become less capacity-limited, the execution times of the synergistic caching system and the L2 system begin to even out. If the level-1 cache is not limited by capacity misses, the performance of both

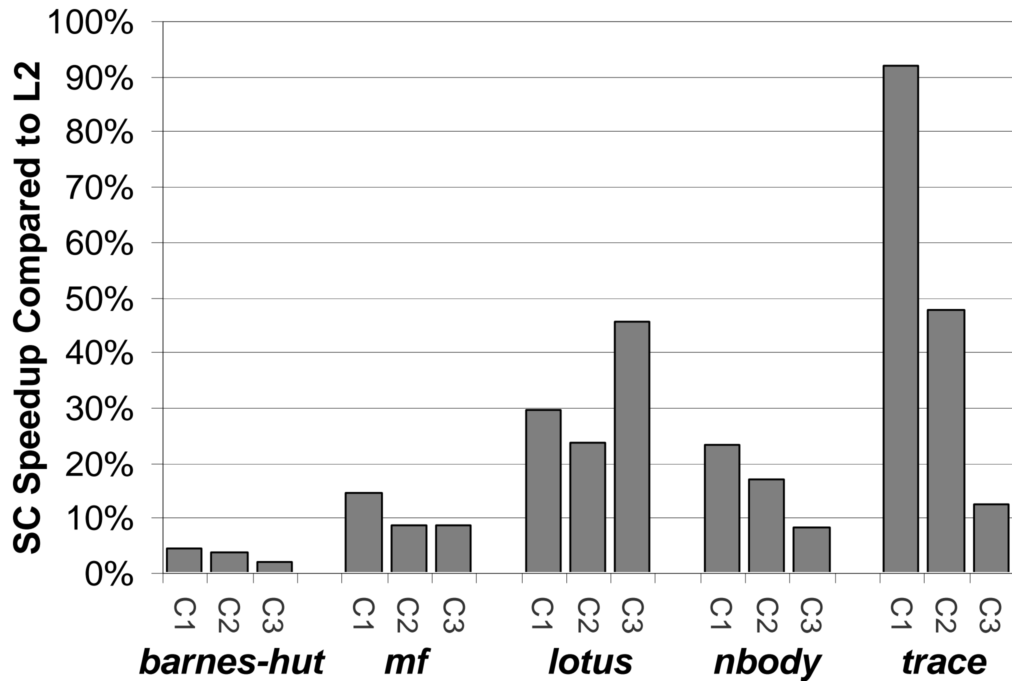

Figure 5.8: Synergistic Caching Execution Speedup as Compared to a Shared-L2 System.

systems is essentially the same. Shared data can be retrieved from the cluster cache or L2 cache, respectively.

The third trend is that for applications with little shared data—*trace* and *nbody*—synergistic caching still performs better. In this case, the added area of the L2 cache is essentially wasted since there is little sharing. The area allocated to the L2 cache would have been more effectively used to increase the capacity of the L1 cache.

While synergistic caching effectively uses cache area even in applications with little sharing, an optimization would be to discontinue cluster requests when few cluster cache hits are detected. This would conserve network and buffer resources and decrease contention at the cache ports.

5.3 Summary

In our comparison of the synergistic caching system with a shared-L2 caching system, we observed that the difference between the systems is most noticeable when the applications are capacity-limited. As L1 cache sizes increase and the applications become less capacity-limited, the performance of both systems becomes comparable.

Under very tight area constraints, the shared-L2 system exacerbates capacity-limited applications and performs worse than the synergistic caching system. In these cases, synergistic caching performs at a more optimal level on the L1 hit rate curve. However, when area is not as limited, the synergistic caching system shows little improvement over the shared-L2 system. In fact, when area is not the limiting constraint, the overhead of increases in the critical path and added complexity overwhelm the benefits of synergistic caching.

Chapter 6

Conclusions and Future Work

In this thesis we introduced synergistic caching as a means of alleviating the processor-memory bottleneck by trading off on-chip bandwidth for area. With the emergence of chip multiprocessor systems, the cost of communication is changing and—as opposed to multi-chip systems—in the on-chip environment, wires are cheap and area is expensive. Synergistic caching takes advantage of this on-chip environment and of sharing in multi-threaded applications to make shared data on neighboring nodes easily accessible.

We showed that synergistic caching is beneficial when chip area is limited and when there are large amounts of sharing in the desired set of applications. When this is the case, chip area can be used for processing power instead of for additional levels in the memory hierarchy – level-2 or level-3 caches – and synergistic caching can act as an effective second-level cache to reduce average memory access time.

The added functionality of synergistic caching requires extra engineering, putting more hardware in the critical path. Specifically, the cost is an increase in the critical path due to the additional multiplexers and the cluster request buffer, and an increase in network traffic. However, as shown in the previous chapters, interconnect bandwidth was not limiting for the benchmarks given. Thus, when the system is limited

by total area constraints and requires large amounts of processing power, synergistic caching proves to be beneficial. In those cases, the area that would have been spent on the L2 cache can be devoted to additional processors. These systems that perform large amounts of processing per memory access can be found in high-end routers that use 100's of simple processors with very small caches.

However, for non-capacity limited applications, we showed that one would be better off using an L2 cache instead of interrupting the neighboring processor. We showed that the performance of the synergistic caching system shows little improvement over a traditional shared level-2 cache for larger L1 cache sizes. In these cases, the additional costs in area, critical path delay, and network traffic overwhelm the benefits of the synergistic caching system.

For further investigation of synergistic caching technology and its implications, we propose three main directions. First, the studies shown in this thesis looked at the kernels of the applications—the parallel portions of the code. However, many parallel applications have large amounts of single-thread start up code and end code. For these serial sections of code, the application thread could make use of the neighboring caches as an effective expansion of its local cache. While neighboring nodes wouldn't actively pull in needed data, as in the traditional synergistic caching system, neighboring nodes could act as victim caches for the running thread. The system would detect that a single thread is running and use the neighboring node cache capacity. In this way, the area devoted to the caching system is completely utilized to increase overall application performance. After the single thread spawns additional threads on neighboring nodes, the system detects that other threads are running and discontinues the use of neighboring nodes as victim caches. The single thread's use of the entire cluster cache may effectively warm-up the neighboring caches, and data on neighboring nodes can still be used by the original thread until they are evicted from the neighboring node's active thread.

The second area of investigation is that of reconfigurability in the synergistic caching system. This was alluded to in Chapter 3 in the discussion about duplication modes. By adding feedback, the system can adapt to different application characteristics and data footprints. In the case of duplication modes, as mentioned in Chapter 3, a counter may be added to detect the number of hits in the L1 cache. If the number of hits dips below a threshold, the duplication mode can change from *beg* to *borrow* or *steal*. Additionally, if an application has little or no sharing, requesting data from the cluster caches will add no benefit. A counter or lookup table could be used to detect when there are few hits in the cluster cache. At that point, requests to the cluster cache can be suspended. This will effect an overall reduction in network demand, buffer usage, and port contention at neighboring caches.

Lastly, investigations could be made into modifying synergistic caching to use technology proposed by Kim et al.[9]. As cache sizes increase, they also become slow. This makes them non-ideal for building L1 caches. However, caches are built using internal banks. As observed by Kim et al., as cache size increases, delay to near banks may be an order of magnitude different from the delay of far banks [9]. Using this understanding to the advantage of the system would allow closer blocks to be used as the effective L1 cache, and farther away blocks to be used as the effective L2 cache. Synergistic caching could then be used in conjunction with these larger cache sizes and, instead of artificially partitioning the L1 and L2 caches, the entire cache can be partitioned dynamically as it is used by the system.

In conclusion, we proposed that one could use the increased bandwidth to neighboring nodes to effectively increase cache size without using additional area. We showed that, for capacity-limited scenarios, synergistic caching takes advantage of the high on-chip bandwidth of CMPs, minimizes the area dedicated to caching, and takes advantage of sharing to improve performance. The cost of doing this is adding

hardware to the critical path. Because of these costs, we found that returns are modest for larger cache sizes, making it beneficial to use synergistic caching only when the system is area constrained, and the user would rather use the area for processing power than for additional levels in the memory hierarchy.

Appendix A

Glossary

Blocking Memory:

A blocking memory system stalls the running thread when the data is not retrieved in a single processor cycle.

Cache:

A fast, small memory system that stores frequently accessed data. The access delay is typically on the order of 1 processor clock cycle.

Capacity-Limited Application:

An application is capacity-limited when the working set does not fit in the level-1 cache. When an application is capacity-limited, it is expressed by capacity misses in the cache. See entry for capacity miss.

Capacity Miss:

A capacity miss in a cache occurs when data is evicted from the cache only to be retrieved again (from main memory) in a later cycle. If the cache had been a large enough, the data would not have been evicted from the cache and could have been supplied by the cache instead of main memory.

Chip Multiprocessor:

A chip multiprocessor (CMP) is a computing system with multiple processing cores on a single chip.

Cluster:

A number of nodes that share data at the L1 cache level of a synergistic caching system.

Cluster Cache:
The conglomerate caches of all of the nodes in the cluster of a synergistic caching system.

Cluster Cache Capacity:
the total number of unique bits in the entire caching system.

Cluster Request Buffer:
A buffer that holds incoming requests, cluster requests, from neighboring nodes. The cluster request buffer only holds cluster requests if the cache port is not available.

Duplication Mode - Beg:
Data requested from a neighboring node are into the requesting node's cache.
Duplication Mode - Borrow:
Data requested from a neighboring node are used by the requesting processor but not copied into the local cache.
Duplication Mode - Steal:
Data requested from a neighboring node are copied into the local cache, and the supplying node's data is invalidated.

Latency Tolerance:
the ability of an application to allow delay in the memory system.

Level-1 Cache:
The level-1 (L1) cache is the first place a processor checks for data. **Also see: Cache**.

Level-2 Cache:
The level-2 (L2) cache is a larger, slower cache than the L1 cache. After missing in a L1 cache, the processor checks the L2 cache.

Load-use Distance:
the time between when data is loaded into a register and when it is actually used in a calculation.

MAT:
The average memory access time (MAT) is the time it takes, on average, to access data.

Multi-threading: Applications written with multi-threading have multiple lists of instructions that may be run independently either on different nodes or on the same node.

Multiprocessor:
A computing system that has multiple processing cores.

Non-blocking Memory:
A thread is stalled only when the register waiting to be loaded with data from memory is actually used, not when the data is first requested.

Outgoing Request Buffer:
A buffer that holds a list of all requests that are still pending for a single node. If a line is pending, and the processor requests the line again, the cache controller checks the outgoing request buffer and, upon a hit in the buffer, the request simply waits for the data to return. Multiple requests to the same data line are avoided.

Overall Hit Rate:
The percentage of memory accesses that are serviced by any system other than main memory.

Private Data:
Data accessed by a single thread in a given application.

Re-Order Buffer:
A buffer held by the processor that re-orders data requests as they are retrieved from memory. Because data delays vary for each data address, the re-order buffer keeps track of which came first and makes sure the processor sees the data in an order consistent with the request ordering.

Shared Data:
Data accessed by more than one thread in a single application.

Multi-threading (MT):
An application that uses many threads running concurrently on a single node.

Write Buffer:
A buffer that holds four kinds of written data: writes issued by the processor, invalidates issued by other processors, cluster replies, and main memory replies. The

data are held in the write buffer only if the cache port isn't available. When a processor requests data, it simultaneously checks the cache and the write buffer. If it hits in either location, the data are returned immediately.

Bibliography

[1] A. Agarwal, R. Simoni, J. L. Hennessy, and M. Horowitz. An evaluation of directory schemes for cache coherence. In *25 Years ISCA: Retrospectives and Reprints*, pages 353–362, 1998.

[2] G. H. Barnes, R. M. Brown, M. Kato, D. J. Kuck, D. L. Slotnick, and R. A. Stokes. The illiac iv computer. In *IEEE Transactions on Computers*, pages 746–757, August 1968.

[3] 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 *Proc. ACM International Symposium on Computer Architecture (ISCA'00)*, pages 282–293, 2000.

[4] J. A. S. Fiske. *Thread Scheduling Mechanisms for Multi-Threaded Parallel Processors*. PhD thesis, Massachusetts Institute of Technology, Boston, 1995.

[5] L. Hammond, B. A. Hubbert, M. Siu, M. K. Prabhu, M. Chen, and K. Olukotun. The Stanford Hydra CMP. *IEEE MICRO*, 20:71–84, March 2000.

[6] J. Hennessy and D. Patterson. *Computer Architecture: A Quantitative Approach*. Morgan Kaufmann Publishers, San Mateo, CA, USA, 1990.

[7] T. Juan, J. J. Navarro, and O. Temam. Data caches for superscalar processors. In *International Conference on Supercomputing*, pages 60–67, 1997.

[8] R. Kalla, B. Sinharoy, and J. Tendler. Simultaneous multi-threading implementation in Power5—IBM's next generation POWER microprocessor. *Hot Chips 15*, 2003.

[9] C. Kim, D. Burger, and S. W. Keckler. An adaptive, non-uniform cache structure for wire-delay dominated on-chip caches. In *Proc. IEEE International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS'02)*, pages 211–222, 2002.

[10] P. Kongetira. A 32-way multithreaded sparc processor. In *Hot Chips 16*, 2004.

[11] 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. L. Hennessy. The stanford FLASH multiprocessor. In *Proc. of the 21th Annual Int'l Symp. on Computer Architecture (ISCA'94)*, pages 302–313, 1994.

[12] J. Laudon and D. Lenoski. The SGI origin: A ccNUMA highly scalable server. In *Proc. of the 24th Annual Int'l Symp. on Computer Architecture (ISCA'97)*, pages 241–251, 1997.

[13] D. Lenoski, J. Laudon, K. Gharachorloo, A. Gupta, and J. Hennessy. The directory-based cache coherence protocol for the dash multiprocessor. In *Proc. of the 17th ACM International Symposium on Computer Architecture (ISCA'90)*, 1990.

[14] H.M. Levy and Jr. R.H. Eckhouse. *Computer Programming and Architecture - the VAX*. Digital Press, Newton, MA, U.S.A., 1989.

[15] C. Liu, A. Sivasubramaniam, and M. Kandemir. Organizing the last line of defense before hitting the memory wall for CMPs. In *Proc. IEEE International Symposium on High Performance Computer Architecture (HPCA'04)*, pages 176–185, 2004.

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

[17] M. M. K. Martin, M. D. Hill, and D.A. Wood. Token coherence: A new framework for shared-memory multiprocessors. *Micro*, 2003.

[18] G. E. Moore. Cramming more components onto integrated circuits. *Electronics*, 38(1), 1965.

[19] M.F. Mudawar. Scalable cache memory design for large-scale smt architectures. In *Proceedings of the 3rd Workshop on Memory Performance Issues*, 2004.

[20] H. L. Muller, P. W. A. Stallard, and D. H. D. Warren. The data diffusion machine with a scalable point-to-point network. Technical Report CSTR-93-17, 1, 1993.

[21] B. A. Nayfeh, L. Hammond, and K. Olukotun. Evaluation of design alternatives for a multiprocessor microprocessor. In *Proc. ACM International Symposium on Computer Architecture (ISCA'96)*, pages 67–77, 1996.

[22] B. A. Nayfeh and K. Olukotun. Exploring the design space for a shared-cache multiprocessor. In *Proc. ACM International Symposium on Computer Architecture (ISCA'94)*, pages 166–175, 1994.

[23] K. Research. KSR technical summary. *Kendall Square Research Corporation Technical Summary*, 1992.

[24] J. A. Rivers, G. S. Tyson, E. S. Davidson, and T. M. Austin. On high-bandwidth data cache design for multi-issue processors. In *International Symposium on Microarchitecture*, pages 46–56, 1997.

[25] K. Sankaralingam, R. Nagarajan, H. Liu, C. Kim, J. Huh, D. Burger, S. W. Keckler, and C. R. Moore. Exploiting ILP, TLP, and DLP with the polymorphous trips architecture. In *Proc. of 30th ACM Internation Symposium on Computer Architecture (ISCA-03)*, pages 422–433, 2003.

[26] P. Shivakumar and N. P. Jouppi. Cacti 3.0: An integrated cache timing, power and area model. *WRL Research Report*, August 2001.

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

[28] L. Spracklen and S. G. Abraham. Chip multithreading: Opportunities and challenges. In *Proceedings of the Eleventh International Symposium on High-Performance Computer Architecture (HPCA-11 2005)*, 2005.

[29] M. B. Taylor, J. Kim, J. Miller, D. Wentzlaff, F. Ghodrat, B. Greenwald, H. Hoffman, P. Johnson, J.-W. Lee, W. Lee, A. Ma, A. Saraf, M. Seneski, N. Shnidman, V. Strumpen, M. Frank, S. Amarasinghe, and A. Agarwal. The RAW microprocessor: A computational fabric for software circuits and general-purpose programs. *Micro*, 2002.

[30] D. M. Tullsen, S. Eggers, and H. M. Levy. Simultaneous multithreading: Maximizing on-chip parallelism. In *Proceedings of the 22nd Annual International Symposium on Computer Architecture*, 1995.

[31] D. H. D. Warren and S. Haridi. Data diffusion machine – a scalable shared virtual memory multiprocessor. In *Fifth Generation Computer Systems*, pages 943–952, 1988.

[32] K. M. Wilson, K. Olukotun, and M. Rosenblum. Increasing cache port efficiency for dynamic superscalar processors. In *Proceedings of the 23rd Annual International Symposium on Computer Architecture (ISCA96)*, pages 147–157, 1996.

[33] K.M. Wilson and K. Olukotun. Designing high bandwidth on-chip caches. In *Proceedings of the 24th Annual International Symposium on Computer Architecture (ISCA97)*, pages 121–132, 1997.

[34] W. A. Wulf and S. A. McKee. Hitting the memory wall: Implications of the obvious. *Computer Architecture News*, 23(1):20–24, 1995.