EXPLORING, DEFINING, AND EXPLOITING RECENT STORE VALUE LOCALITY

by

Kevin M. Lepak

A dissertation submitted in partial fulfillment of

the requirements for the degree of

Doctor of Philosophy

(Electrical Engineering)

at the

UNIVERSITY OF WISCONSIN-MADISON

2003

Abstract

This thesis is motivated by the growing differential between main memory and microprocessor core performance. Increased integration, enabled by Moore's law, has provided a substantial compound improvement in core performance. Integration has benefitted main memory latency less significantly, leading to an expanding *memory-gap*. Furthermore, in multiprocessors, increasing integration has allowed enlarging on-chip cache structures to continue reducing capacity and conflict misses; however, communication misses still remain, limiting performance of multithreaded workloads. Locality in both temporal and spatial dimensions has been exploited historically by computer architects to improve memory system performance.

Recently, a new locality dimension has emerged unveiling additional potential for performance improvement. Value locality describes a program behavior phenomenon in which values recur in programs. Many researchers have examined value locality as a means to improve memory system performance. However, most research has focused on predicting load values, as it is believed that loads are latency critical. In contrast, conventional wisdom says stores are not latency critical and need only be buffered and forwarded for acceptable performance.

In this thesis, we show that stores should be examined as a means of improving memory performance for both uniprocessors and multiprocessors and that stores exhibit significant value locality. For example, approximately 40% of stores are *update silent*; they write the same value which already exists at the memory location, thus contributing no change in system state. We show numerous methods of exploiting store value locality to increase performance. In uniprocessors, we detail improvements in core efficiency; in multiprocessors, significant reductions in communication between processors.

We focus predominantly on multiprocessors, making a fundamental contribution in redefining multiprocessor sharing to consider two dimensions of store value locality. Furthermore, we describe both speculative and non-speculative methods which achieve substantial performance benefit by exploiting store value locality in both scientific and commercial workloads. Many of our proposals can be integrated into existing microprocessor designs with coherence protocol changes, while others rely on existing coherence

mechanisms to reap tangible benefit. We perform a detailed performance evaluation, using full-system, execution-driven, simulation to show the merits of different designs.

Acknowledgements

I would like to thank my parents, Ken Lepak and Carol Lepak for their constant support throughout my educational career. I owe my strong work-ethic and drive to their influence and I am truly grateful for that. I thank my mom especially for her willingness to encourage my interest in technology and engineering at a very young age. I also thank my siblings, Kurt, Ann, and Julie, for their support and honor the memory of Kristopher.

My advisor, Prof. Mikko Lipasti has influenced me in numerous ways. He taught me about academic engineering research and how to ask the right questions. I also thank him for his academic and financial support throughout my academic career. In addition, I am a truly a better person having worked with Mikko. I learned through example how to be true to yourself, those important to you, and your beliefs, even under exceptional circumstances. I additionally thank the members of my thesis committee, Prof. Jim Smith, Prof. Mark Hill, Prof. David Wood, and Prof. Guri Sohi for their feedback on my thesis research, this thesis document, and stimulating discussions throughout my graduate tenure. Additionally, I thank Prof. Jim Goodman and Prof. Ras Bodik who assisted with my preliminary proposal and illuminated avenues to be explored in this thesis.

On a collaborative level, I thank Jason Cantin for stimulating coherence validation discussion and PHARMsim visualization tools, Ravi Rajwar for detailed explanations regarding SLE, and all members of the PHARM research group for their technical assistance and contribution to shared research infrastructure during my tenure. I especially thank Trey Cain for his substantial PHARMsim development effort, and Ilhyun Kim for many stimulating discussions. I also thank Ilhyun and Gordie Bell for cooperating with me on some research related to this thesis. I thank Jay Pickett, Mitch Alsup, and Ben Sander

from Advanced Micro Devices who taught me much about real-life microprocessor design and architecture design/evaluation during my long internship between finishing my M.S. and returning to Wisconsin to finish my Ph.D.

On a personal level, I am lucky to have worked with Trey, Gordie, and Ilhyun. Being Mikko's first students, we have been through a lot together and at times it feels like we have accomplished herculean tasks. I thank Brian Mestan, Tim Snyder, Chris Belmas, and Greg Zagorski for always being true friends to whom I could relate my feelings. I thank Milo Martin for encouraging me to attend graduate school in the first place and for our interactions throughout my graduate career. I especially thank Katie Hubbard who has seen my metamorphosis throughout graduate school, was there through some of the most turbulent times, and reminded me who I really was during some of the toughest moments.

I thank Greg, Tim, Razvan Cheveresan, and Matt Ramsay for providing welcome diversions from the rigors of graduate school. I thank all the members of MAUL ultimate frisbee for accepting me onto the field of honor twice a week for the last five years, providing much needed exercise and stress relief, although I never could throw . . . at all.

Finally, I thank the Department of Electrical Engineering staff for their assistance with administrative tasks throughout my graduate career. I especially thank Bruce Orchard, the network administrator, who was always willing to let us do whatever we needed to get the job done—and help us put the pieces back together when things went horribly awry.

Table of Contents

List of Figures

List of Tables

Chapter 1

Introduction

Computers are pervasive in everyday life. We are continually surrounded by microprocessors and microprocessor systems; from the computer which sits on the desktop in places of business, to the server which powers e-commerce, to the multitude of embedded microprocessors we encounter every day inside of household appliances, automobiles, and entertainment devices. Computer architects have exploited greater transistor budgets (afforded by semiconductor manufacturing improvements, i.e. “Moore’s Law” [84]) to continually improve performance and enable tighter levels of integration within computer systems.

However, this trend of increased integration has benefitted different parts of the modern computer system at different rates. Due to increased demand (from users) for both greater processor power and memory capacity, a disparity has developed between the rate at which microprocessor cores can perform computation and the rate at which memory systems can supply data to the core. There are many reasons for this trend, some related to fundamental physical phenomena (making a memory both large and fast is difficult or impossible), others to economics within the semiconductor industry (a desire to commoditize memory and achieve economies of scale). This trend has been quantified by Hennessy and Patterson [46] and has been termed the “memory-gap”. We reproduce their data here in Figure 1-1.

It is obvious from the figure that increasing integration has benefitted microprocessor core performance at a greater compound rate than it has benefitted main memory latency. This growing gap between processor and memory speeds indicates that memory

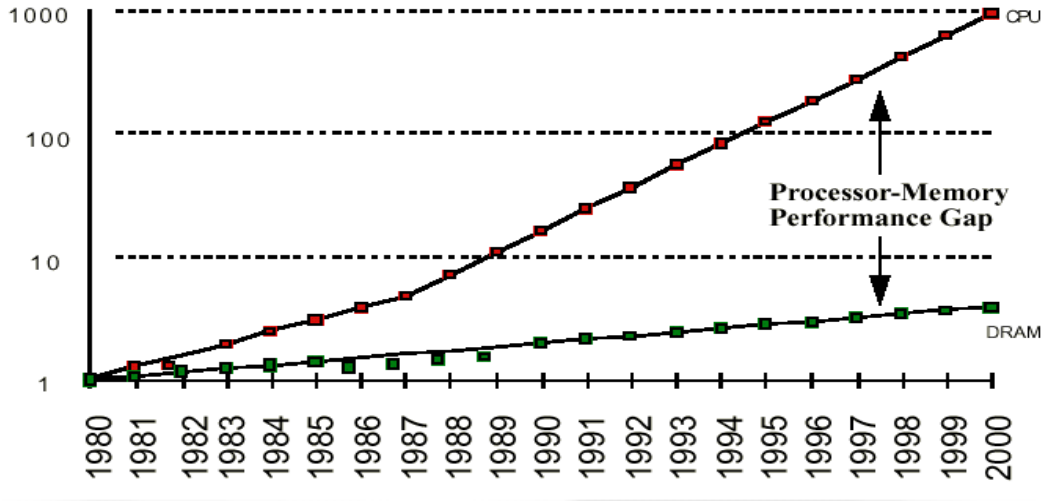

FIGURE 1-1. The Growing Gap Between Processor Performance and Memory Latency. The figure shows the compound growth in processor core ("CPU") performance and reduction in main memory latency from 1980 through 2000. Note the logarithmic scale of the y-axis. Taken from Hennessy and Patterson [46].

system performance will only become more important in future processors due to Amdahl's Law [8], motivating the research of this thesis in improving memory system performance. Historically, architects have focused on improving the latency of memory loads (reads) as opposed to stores (writes) because loads are both more prominent as a fraction of dynamic instructions executed and it is traditionally believed that stores are not on the critical path of execution because they can be buffered and forwarded.

Indeed, loads constitute the dominant fraction of memory operations. For a PowerPC-like RISC instruction set architecture, memory operations contribute approximately 40% of dynamic instructions, with 25% being loads and 15% being stores [4]; we have observed similar proportions in our workloads. Although store operations contribute a smaller percentage of memory operations in general, handling each dynamic store is difficult. Within a processor core, complicated load-store queue structures are required to handle memory aliases, first level caches must be specially designed to handle the destructive nature of stores, and true dependences through memory, i.e. store-to-load forwarding, can

cause load operations to be delayed. We will further discuss these issues in Section 2.1; obviously, store performance is important in optimizing memory systems even though stores contribute a smaller fraction of dynamic memory operations.

1.1 Performance Impact of Communication Misses in Multiprocessor Systems

Most small/medium-scale multiprocessor systems are cache-coherent, shared-memory systems which provide the programming abstraction of a single, coherent, memory image shared among all processors in the system. This abstraction is useful because it allows all data in memory to be shared among processors automatically without explicit messages from the programmer [31].

Creating a high-performance memory system for such machines is complicated for many reasons, relating to both performance and correctness. A fundamental correctness consideration in shared-memory systems is maintaining a coherent shared memory image among processors. Stores performed by a processor must become visible to other processors in the system through some mechanism; in multiprocessor systems with caching store visibility is a key aspect of what we commonly call the cache coherence problem.

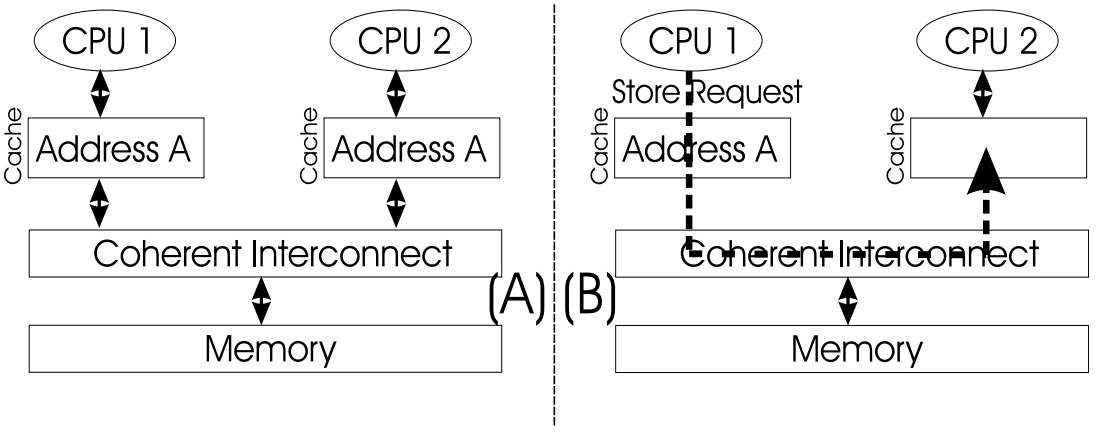

FIGURE 1-2. Basic Cache-Coherent Shared-Memory System Operation. The figure indicates an initial state where Address A is cached by both CPU 1 and CPU 2 (A). In order to propagate new values to other processors and maintain a coherent view of memory, stores send a request on the coherent interconnect invalidating remote cached copies of Address A (B).

We illustrate the problem and a potential solution in Figure 1-2. In part (A) of the figure, we show a two-processor system with a single cache for each processor. Both caches have a copy of the data stored at Address A in their local cache to enable low-latency access to the data. However, if each processor has its own copy of the data stored at Address A, how does the system ensure that a write to that location is made visible to other processors? A commonly implemented solution is illustrated in part (B) of the figure. When a store operation is to be performed (by CPU 1), it sends a request out to the other processors in the system (CPU 2) indicating that they should invalidate their copy of Address A. Once the invalidation is carried out, CPU 1 is the only processor with a copy of Address A, hence it is safe to store to it. Another processor observes this new write if it later accesses Address A; it experiences a cache miss, sends a request on the coherent interconnect, and obtains the updated data from CPU 1. Cache misses induced by sharing of data between processors are variously called *communication*, *coherence*, or *dirty* misses in the literature [31]. Such misses are considered fundamental in multi-threaded program execution because true sharing of data is occurring between processors (for a detailed classification of different types of cache misses in uniprocessors, see Hill's thesis [48]; a discussion of communication misses is available in Culler and Singh [31]).

Researchers have reported that about one-half of all off-chip memory accesses are due to communication misses in current generation systems [9] [75]. As technology trends continue to increase the number of processors in a single system, the probability of communication between processors increases. Furthermore, these same technology trends also allow increasing the cacheable memory space within the system, eliminating many capacity/conflict misses. Therefore, we expect communication misses to maintain or grow their

share of impact on memory system performance in the future. Obviously, eliminating such misses will improve system performance.

Note that communication misses exist because of the need to communicate new values through memory; these new values are created by store operations. Therefore, it should be apparent that store operations, even though they contribute a small fraction of dynamic instructions, are costly in terms of their impact on microprocessor core design and their detrimental effect on system performance in multiprocessor systems. Traditionally, architects have exploited locality in the temporal and spatial directions for memory references in order to improve performance of costly memory operations [46].

1.2 Exploiting Value Locality to Improve Memory System Performance

Recently, a new locality dimension has emerged which we can exploit as computer architects. Value locality has been widely studied since the seminal work was presented by Lipasti [73] [72] [81]. Value locality describes the program behavior phenomenon that values tend to recur throughout program execution or are trivially predictable. Previous to this seminal work, architects have primarily focused on improving dataflow between dependent operations to improve execution performance, without regard for the values actually being communicated through the dataflow. However, it is the values themselves which are required to perform computations; therefore, through exploiting locality of values we can potentially improve performance, even beyond the previous *dataflow-limit* rate of execution [73].

Prior studies examining value locality of memory operations are principally concerned with predicting load values, which can be thought of as accelerating value consumption. Surprisingly, value locality has been explored very little for store operations,

which is in effect, memory value production. Accelerating production of values also accelerates consumption, but viewing the problem from this different angle leads to novel methods to improve performance and observations of program behavior.

In this thesis, we explore the value locality of stores. If this locality exists, intuitively we may be able to exploit it to simplify and accelerate memory communication. Understanding the value behavior of stores—the only way true, persistent, program output can be communicated—may also illuminate parts of the fundamental computation performed in machines.

Throughout the next few sections, we briefly introduce the dimensions of store value locality explored in this thesis.

Table 1-1: Dynamic Update Silent Stores for Uniprocessor and Multiprocessor Benchmarks.

Benchmark	Description	Update Silent Stores (PPC/SS)
go	SPEC95 game	38%/27%
m88ksim	SPEC95 simulator	68%/62%
gcc	SPEC95 compiler	53%/46%
compress	SPEC95 compression	42%/39%
li	SPEC95 lisp interpreter	34%/20%
ijpeg	SPEC95 image compression	43%/33%
perl	SPEC95 language interpreter	49%/36%
vortex	SPEC95 object database	64%/55%
tomcatv	SPEC95 vectorized mesh generation	47%/33%
swim	SPEC95 shallow water equations	34%/26%
mgrid	SPEC95 3D potential field	23%/7%
applu	SPEC95 partial differential equations	37%/35%
apsi	SPEC95 weather prediction	21%/25%
fpppp	SPEC95 Gaussian quantum chemistry	15%/15%
wave5	SPEC95 Maxwell's equations	25%/22%
gzip	SPEC2000 file compression	28%/8%
vpr	SPEC2000 FPGA circuit placement and routing	44%/35%
mcf	SPEC2000 combinatorial optimization	65%/64%
parser	SPEC2000 word processing	52%/35%
bzip2	SPEC2000 file compression	8%/21%
barnes	4proc SPLASH-2 N-body simulation [107]	39%
ocean	4proc SPLASH-2 Ocean simulation	22%
radiosity	4proc SPLASH-2 Light Interaction simulation	42%
raytrace	4proc SPLASH-2 Raytrace with teapot input set	38%
tpc-h	4proc decision support (TPC-H query 12) [106]	73%
tpc-w	4proc TPC-W shopping mix [18]	68%
specjbb	4proc SPECJBB2000 Java benchmark [18]	35%
specweb	4proc SPECWEB99 benchmark [18]	37%

The first type of store value locality (SVL) we explore in this thesis is *update silent stores*. We call a store update silent if it writes the same value which already exists at the memory location of interest (and hence creates no update/change in system state). Table 1-1 indicates the percentage of dynamic update silent stores for a sample of uniprocessor and multiprocessor benchmarks; we show data for all benchmarks which have been setup and run in our simulation environments. Two ISAs (Simplescalar PISA [14] and

PowerPC [56]) are shown for the uniprocessor benchmarks while the multiprocessor benchmarks are studied under PowerPC only as we do not have multiprocessor versions of these benchmarks for SimpleScalar.[1] The key point is that significant store value locality exists. For the programs in the table, a range of 7% to 73% of all dynamic stores are update silent, across single-threaded and multi-threaded programs and ISAs. This result is surprising—on average approximately 40% of memory writes are not contributing new information to the state of the system. This thesis will explore causes for this phenomenon, important ISA and program considerations, and methods of exploiting update silence for improved performance in both uniprocessor (Chapter 2) and multiprocessor (Chapter 3, Chapter 4) systems. In previous work, we referred to update silent stores as simply *silent stores*, however to differentiate between update silent stores and other types of store value locality which we illuminated subsequently (to be introduced in the next sections) we encourage the usage of the term update silent store throughout future literature to eliminate ambiguity. The term silent store (with no modifier) should be interpreted to mean any form of store value locality. For brevity, the term silent store may be used without modification as long as the store value locality of interest is unambiguous or all types of store value locality are implied.

1.4 Temporally Silent Stores

Update silent stores do not change the system state. A natural extension is to consider stores which do change the state (and so are not update silent as described previously), but do in fact change the state back to some previous value. We call stores writing such values *temporally silent stores*. In Table 1-2 we show the percentage of stores which

1. We discuss differences observed between SimpleScalar and PowerPC throughout Chapter 2, and specifically in Section 2.6.

.

are temporally silent[2] across the 4-processor benchmarks explored in depth in this thesis.

Although we observe that the percentage of dynamic temporally silent stores is significantly less than update silent stores, we find that exploiting these temporally silent stores in multiprocessor systems can greatly reduce communication misses. On average, 40% of communication misses in our commercial workloads can be removed by exploiting temporal silence. We redefine multiprocessor sharing to correctly consider temporal silence in Chapter 3. Furthermore, we explore the phenomenon of temporal silence in detail through program-level characterization and also design methods to eliminate communication misses caused by temporally silent stores in Chapter 5.

Table 1-2: Percentage of Temporally Silent Stores for 4-Processor Workloads. Instructions are measured excluding the operating system idle loop. *Temporally silent stores* are measured with respect to the previous globally visible version of the cache line (MESTI exploitable) as described in Lepak et al. [70].

	Instr.	Stores	US Stores	TS Stores	Description
barnes	1.76B	304M	117M (39%)	3.4M (1.1%)	SPLASH-2 N-body simulation (8K particles)
ocean	561M	36M	8.0M (22%)	1.8M (5.4%)	SPLASH-2 Ocean simulation (258x258)
radiosity	2.43B	326M	138M (42%)	1.1M (0.34%)	SPLASH-2 Light Interaction application (-room -ae 5000.0 -en 0.050 -bf 0.10)
raytrace	414M	45M	17M (38%)	1.2M (2.6%)	SPLASH-2 Raytracing application (teapot)
specweb	4.60B	624M	233M (37%)	23M (3.7%)	Commercial Web-Serving application
specjbb	3.58B	431M	152M (35%)	22M (5.1%)	Commercial Server-Side Java application
tpc-h	5.05B	551M	401M (73%)	19M (3.4%)	Commercial decision support (query 12)
tpc-w	1.41B	340M	231M (68%)	9.2M (2.7%)	Commercial Web-Based OLTP application

1.5 Thesis Overview and Summary of Contributions

Our examination of store value locality begins with a detailed exploration of update silent stores in uniprocessor systems—exploring architectural performance improvement possibilities, efficient methods of detecting them, and a brief summary of characterization/program analysis studies which provide insight (Chapter 2). We will then explore how multiprocessor sharing can be redefined to consider store value locality.

2. The definition of temporal silence used throughout this thesis is more restrictive than described here. For now, we only seek to indicate the general idea; details are given in Section 5.1.

These studies indicate potential to improve multiprocessor system performance by eliminating communication misses and address traffic (Chapter 3). With terminology in place, we will then discuss exploiting update silent stores and temporally silent stores in multiprocessors in detail (Chapter 4, Chapter 5), providing extensive characterization and performance projections to guide the methods chosen to exploit store value locality. Having characterized store value locality in detail, we present an evaluation in snoop-based multiprocessors with full-system, execution-driven simulation (Chapter 6). Finally, we summarize and conclude (Chapter 7).

This thesis constitutes a detailed examination of store value locality, exploiting it in uniprocessors and multiprocessor systems using both scientific and commercial workloads. It contributes the following to the state of the art:

- Explores, in depth, the locality of memory value production (in contrast to extensively explored memory value consumption), providing an additional perspective from which to view memory value patterns.
- Demonstrates that significant store value locality exists, and from which viewpoints it can be most effectively exploited (as described briefly in Section 1.3 and Section 1.4), contributing a new direction for value locality-related research.
- Illustrates novel microarchitectural methods for detecting update silent stores and integrating these methods into processor cores for uniprocessor and multiprocessor performance enhancement.
- Redefines multiprocessor sharing to account for store value locality, illuminating substantial opportunity to eliminate communication misses, providing a new class of avoidable sharing misses previously unexplored.

- Illustrates novel microarchitectural methods for detecting temporal silence and integrating these methods (speculatively and non-speculatively) into both processor cores and system architectures at varying levels of difficulty to substantially reduce communication misses.
- Contributes an understanding of *critical update/temporally silent stores* and how these interact with cache-coherent multiprocessor systems within the microprocessor core, in the coherence interconnect, and the data interconnect.
- Characterizes store value locality in multithreaded programs for both scientific and commercial workloads, allowing a program-level understanding of why store value locality exists. These studies illuminate new aspects of communication patterns and the nature of computation performed in programs.

Chapter 2

Exploiting Update Silence in Uniprocessors

In Chapter 1 we introduced the concepts of update silent stores and temporally silent stores. In Section 1.3, we showed that the fraction of dynamic update silent stores is substantial across ISAs, compilation technologies, and varying types of workloads. Obviously, update silence pervades program execution and we should explore its utility to improve performance. In this chapter, we focus on exploiting update silence in a uniprocessor system to improve memory performance. We discuss two distinct types of memory hierarchies (write-back and write-through) and how update silence can be effectively exploited in each type of hierarchy. We also explore core load/store handling modifications that exploit update silence. We then give a brief survey of techniques for exploiting temporal silence in uniprocessors and mechanisms of exploiting it for further core memory hierarchy improvements. We conclude with a summary of the results and a survey of related work.

2.1 Motivation and Background

Modern uniprocessor load/store handling units are normally complex CAM (content-addressable memory), priority-associative, structures to preserve uniprocessor memory order. These structures allow common optimizations such as store to load forwarding, hoisting of loads past unresolved stores, detection of consistency model violations, and other in-core memory performance enhancements. Often, the appropriate trade-off to ease complexity of these structures is to limit the number of in-flight store operations [25, 30, 55, 83]. Eliminating stores which communicate no meaningful change in program state may simplify these structures, and even improve dataflow performance in machines which

detect and stall on true store-to-load dependences by allowing loads to advance around stores which are either predicted to be, or are verified to be, update silent [111]. Reducing the number of cache write ports is also possible and is desirable because special consideration is necessary to support multiple writes per clock cycle. Generally, handling multiple writes requires explicit multi-porting of the SRAM cell or multi-banking, which either complicates the array design or scheduling in the processor core for bank-conflict resolution. Very often, the trade-off chosen in implementations is to constrain the number of stores which can be performed on each processor cycle. Furthermore, due to first level cache designs emphasizing minimum load latency, stores require special handling or multiple trips down the data cache pipeline to determine cache presence before a write can actually be performed into the array. Again, removing unnecessary stores may ease array complexity and reduce memory system occupancy. Finally, eliminating store operations can reduce cache line writebacks/write-throughs and decrease buffering requirements and bandwidth normally used to handle these events.

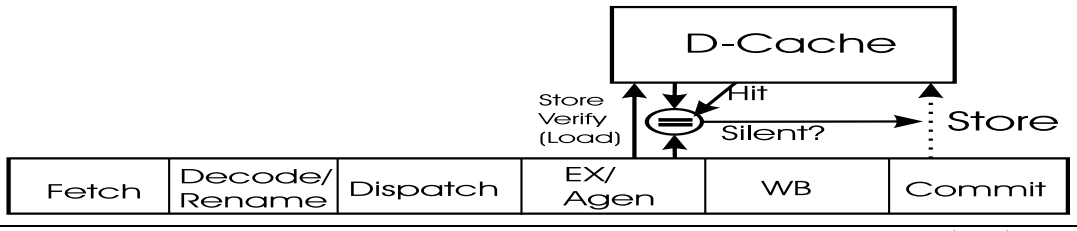

FIGURE 2-1. Standard Store Verify/Suppression Pipeline Diagram. A store operation is converted into a set of load, compare, and conditional-store operations to facilitate update silent store suppression.

2.2 A Simple Microarchitectural Method to Exploit Update Silence

In these uniprocessor studies, we have used a heavily modified version of SimpleScalar 3.0 [14] to study efficient methods of detecting and exploiting update silent stores (detailed machine configuration is given in Section 2.3.4). Before presenting perfor-

mance data, we describe how to detect update silent stores.

2.2.1 Naive Update Silent Store Suppression

We call removal of update silent stores from the dynamic instruction stream *update silent store suppression*. Update silent store suppression describes the overall process of eliminating the write side-effects of an update silent store; *store verification* refers to the subtask of detecting that a store is update silent. A simple way to suppress update silent stores is to convert each dynamic store operation into three operations—a load, compare, and conditional store depending on the outcome of the comparison. This method achieves the subtask of store verification by explicitly performing a load operation to verify the store is update silent; suppression is realized by avoiding the actual write operation at commit for update silent stores. A simplified pipeline diagram of this approach is shown in Figure 2-1.

Of course, a major drawback of this simple approach is that it converts every dynamic store into multiple operations. Minimally, two operations are required (load and compare) when the store is update silent; up to three (load, compare, and store) when the store is not update silent. This has the obvious effect of increasing cache port pressure and power consumption when a store is not update silent (because of two cache accesses), but can improve performance when a store is update silent as discussed in Section 2.1 for microarchitectures which have a high cost for stores. We explore the relative cost of store operations for different architectures, with special consideration for soft-error tolerance, in detail in Section 2.3.

2.3 Advanced Methods For Detecting Update Silence

We can enable update silent store suppression more practically by reducing the

cost of store verification. We explore many such methods in this section. All of these methods can be implemented at a substantially reduced cost as compared to naive store verifies.

2.3.1 Read Port Stealing

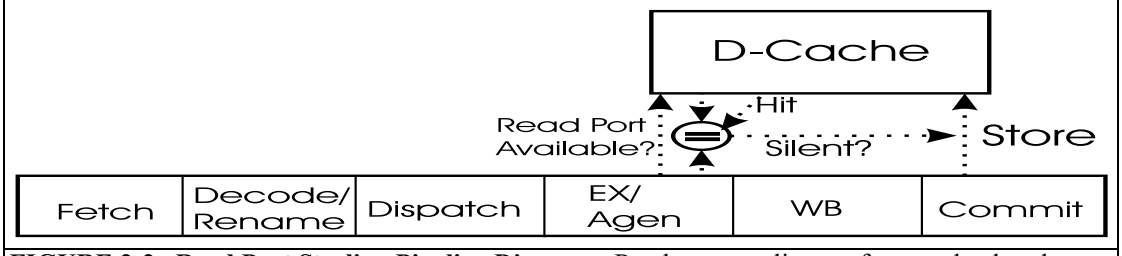

FIGURE 2-2. Read Port Stealing Pipeline Diagram. Read port stealing performs a load and compare only if a cache port is idle.

It is well known that programs are non-uniform in the usage of system resources. Therefore, in many cases, some available idle resources can be used for other purposes. We propose an additional use of idle resources; namely, exploiting free cache read ports to implement store verifies. This mechanism is a simple extension of the standard store verify explained in Section 2.2.1. Since stores must commit in order, it is possible that due to a pipeline stall a store can wait in the LSQ for a long period of time before it completes. If a load port becomes free while the store is waiting to commit, the load port can be used to perform a store verify operation. Because the resources used for verification are idle and available, these store verifies are low-cost. If a load port never becomes available before the store is ready to commit, the pipeline does not attempt to suppress the store and handles it as if it is non-silent.

Relative to naive store verifies, this method has the benefit of not delaying execution of load operations due to resource conflicts. However, it can create additional instruction scheduling difficulties because the policy for issuing a store verify is dependent on

resource usage and not just program order or another static scheduling policy. In machines with unified load/store schedulers such contention for finite resources must already be handled, however, with separate load/store schedulers coordinating access to the memory read ports to ensure collision avoidance between loads and store verifies may prove more difficult. We assume a single load/store scheduler throughout this thesis. We note that both memory scheduler types are common throughout commercial processors, indicating that assuming a single load/store scheduler is realistic. The read port stealing technique is shown in Figure 2-2.

2.3.2 Load/Store Queue

In order to obtain high performance, many processors implement aggressive memory systems which require load/store queues (LSQs) to perform store to load forwarding, monitor speculative load operations which may be violations of the architected consistency model, and other in-core memory optimizations. Since the LSQ provides a window for temporally local references in program order and further enables elaborate memory disambiguation and value forwarding mechanisms to honor program order memory dependences for aggressive processor designs, we can exploit the hardware within the LSQ to obtain reduced cost store verification as well.

2.3.2.1 Temporal Locality in the LSQ

Store to load forwarding between uncommitted loads and stores is a commonly implemented optimization in modern microprocessors. If store forwarding is implemented, we can extend it to suppress later stores to the same address as an earlier store in the LSQ (WAW dependence through memory). We may be able to do so without using a cache read port in some architectures (by serializing store verification's access to the LSQ

and the memory hierarchy), hence making the suppression low-cost.

In a similar fashion, we can also suppress stores to memory addresses for which an outstanding load exists in the LSQ. This is possible because the cache access for the load will be performed, obtaining the data value for the store verify. In some sense, we can consider the store verify for the store to be "piggy-backed" on the explicit load operation to the same memory address (WAR dependence). Note that this optimization is also possible for loads which occur later in program order, which generally would have their load value forwarded from the previous store we're trying to suppress. This is possible because the usage of the cache port is usually scheduled before it is known whether the value will be forwarded from an earlier store in the LSQ [25], [105]. Therefore, since the load has been scheduled for cache access anyway, the load can still be performed at no cost. Hence, the store verify is again low-cost in the case of a RAW memory dependence.

2.3.2.2 Spatial Locality in the LSQ

In a similar fashion, we can expand the scope of suppressible stores within the LSQ to addresses that inhabit the same cache line. Given that L1 data caches are on-chip, obtaining wide access to these caches is relatively easy. Therefore, each memory operation can read an entire cache line on any reference because of the high bandwidth available from the L1 cache. Assuming that a memory access reads the entire line from the cache into a *LSQ cache* (shown in Figure 2-3), the spatially local data can be used to perform additional suppression.

In the case of a WAW dependence, a previous store to the line reads the line into the LSQ cache, and all subsequent stores to that line can be verified from the LSQ cache. In the case of a WAR and RAW dependences, a similar process occurs—the load opera-

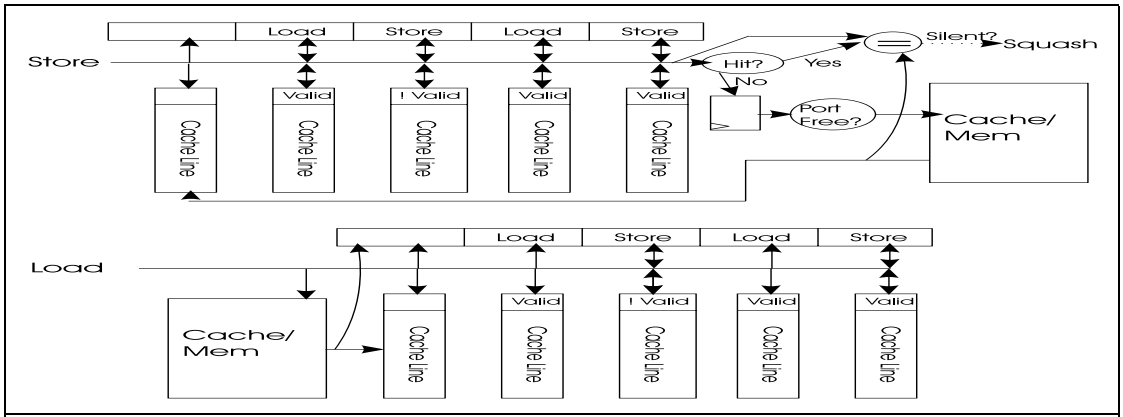

FIGURE 2-3. Block Level LSQ Cache Design.. The temporal and spatial LSQ suppression operations, data allocation, and store forwarding are illustrated for memory operations.

tion allocates the line in the LSQ cache, and stores to the same line are verified from it. We will show in Section 2.4.2.2 that a small LSQ cache is especially effective in the case of WAW dependences.

The LSQ cache is similar to the *write cache* proposed by Jouppi [52], except it contains entire cache lines as opposed to 8 byte quantities and it buffers both load-allocated and store-allocated lines. Also note that since issuing stores is generally not as time critical as issuing loads (because the stores can be buffered at commit) the lookup in the LSQ cache and the access to the memory system (shown in Figure 2-3) can be serialized to avoid unnecessary usage of the data cache port. We can also exploit read port stealing (Section 2.3.1) and only read data for stores into the LSQ cache if a memory read port is available. With read port stealing, a separate valid bit for both the LSQ cache line data and for the entries in the LSQ themselves (shown in the Figure 2-3) is needed because a store may fail to acquire a free read port, leaving the data invalid. When an access allocates a line into the LSQ cache, stores already present in the LSQ can be verified with the newly allocated data, but this may add complexity to the LSQ and LSQ cache for additional data paths. We will discuss this further in Section 2.4.2.2.

If LSQ cache entries are a logical extension of the existing LSQ,[1] explicit tags and

dirty bits for LSQ cache entries are unneeded because all necessary address and dirty value forwarding is already available in the LSQ entries for store-forwarding. In the case of a weakly ordered memory consistency model, it is sufficient for correctness to flush the LSQ cache on memory barriers (this is most likely very effective because of the small LSQ cache size) and avoid snooping it for invalidates. In more strict consistency models, snooping the LSQ may already be required to detect consistency model violations, so snooping the LSQ cache as well adds no additional complexity [110].

The benefits of store verification using the LSQ relative to standard store verifies are apparent. No additional cache access is required for the load portion of the store verify.

2.3.3 ECC Update Silent Store Suppression

We now explore methods of update silent store verification and store suppression with specific consideration of mechanisms for handling soft errors, which are becoming more prevalent as processor cores target implementation technologies with smaller feature sizes. We begin with a brief review of ECC as it relates to store verification, and then describe how update silent store suppression can both be easily implemented, and provide performance benefit, in different ECC architectures.

2.3.3.1 ECC in Modern Microarchitectures

With soft errors in modern microprocessors becoming a larger concern as we move to deeper sub-micron fabrication technologies and higher reliability systems [100, 28, 79, 101, 112, 38], microprocessor designers are protecting the areas of a chip which are most densely packed with transistors, e.g. caches, memories, etc., against random alpha-particles and other causes of soft errors. Error checking and correcting (ECC) codes are a very

1. i.e. entries in the LSQ cache are operated in lock-step with the entries in the LSQ such that when an entry leaves the LSQ its LSQ cache line is also deallocated.

common method for protection against soft errors. ECC using various encoding schemes (we focus on the SEC-DED variety of Hamming based codes [11, 97], but the comments made here apply more generally) requires some number of data bits and check bits to enable the correction of errors. No simple closed form function expresses the number of check bits required for a given number of data bits; however we show the number of check bits required for common data sizes and ECC-word sizes in Table 2-1.

Table 2-1: Data Word Sizes, ECC-Check Bits, and Overhead for ECC Encoding.

Data-word Size (bits)	ECC Check Size (bits)	ECC-word Size (bits)	ECC Check Bit Overhead
8	5	13	62.5%
16	6	22	37.5%
32	7	39	21.9%
64	8	72	12.5%
128	9	137	7.0%
256	10	266	3.9%

There is an obvious trade-off between the granularity on which ECC is kept (data-word size) and the overhead of the check bits. In the case of 13 bit ECC-words (8 data bits), there is a 62% increase in storage space as overhead for ECC. For progressively larger ECC-words, the overhead is reduced—down to 3.9% in the case of 266 bit ECC-words (256 data bits). However, this lower overhead does not come without penalty. Only a single bit error can be corrected and a double bit error detected within the entire ECC-word. Of course, as ECC-word size increases, the probability of multiple errors within a word increases, so ECC is less effective for larger words and a design compromise must be reached. In general, fairly large ECC-word sizes are chosen to minimize overhead and obtain acceptable error coverage. In many modern microprocessors and system busses, 64 bit data-word ECC or larger is used for ease of implementation and because of the configuration of memory systems [26, 28]. As a point of reference, the Alpha 21264 [55] and the

PowerPC RS64-III [13] implement L1 data cache ECC on quadword (64 bit) data quanti- 22
ties.

The check bits for a data-word are generated when a value is stored into the cache and compared when the value is later read (more detail in Section 2.3.3.2). In order to generate correct check bits, all bits in the ECC-word must be available as input to the ECC generation logic. Therefore, if a write operation that is either improperly aligned on ECC-word boundaries, or is a sub-ECC-word sized write, is to be performed, the rest of the original ECC-word stored at the location must be fetched, the changes (from the current write) merged, the new check bits calculated, and the new ECC-word can be stored.

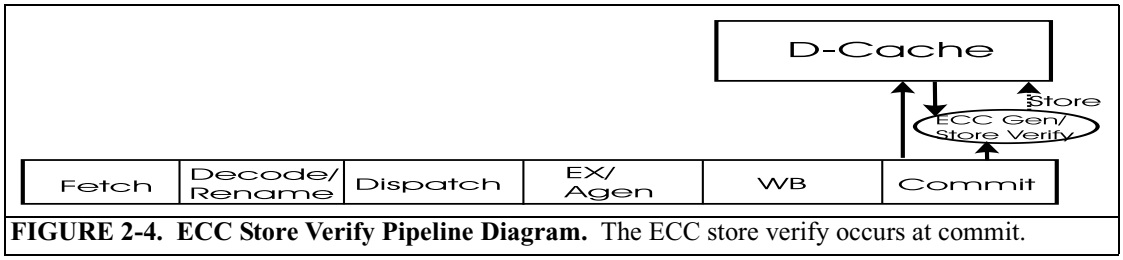

FIGURE 2-4. ECC Store Verify Pipeline Diagram. The ECC store verify occurs at commit.

We can see that in many cases the store operation into an ECC-protected cache really consists of four operations: read original ECC-word, store merge, ECC check bit generation, and new ECC-word store. This realization illuminates the possibility of a new type of low cost update silent store suppression. Since the original ECC-word is read anyway, a comparison of the new store value to the original value can be performed allowing suppression of the update silent stores.

In comparison to naive store verifies (Section 2.2.1), we can see that store verifies carried out in ECC logic require no explicit load operation, but rather can simply be performed at commit, as illustrated in Figure 2-4. The drawbacks of this approach are that a store is suppressed relatively late in the pipeline (at commit instead of during the execute

stage) so it may not reduce pressure on write buffers; it cannot be removed early from the LSQ; and finally that it cannot capture ECC-word-aligned stores.

Now that we have reviewed ECC-coding and the opportunity it presents for reduced cost store verification, we show how such a method can be implemented in modern microarchitectures with two case studies. We also discuss additional methods for providing ECC other than explicit coding protection on the L1 data cache and update silent store suppression in these ECC architectures.

2.3.3.2 L1 Data Cache with ECC

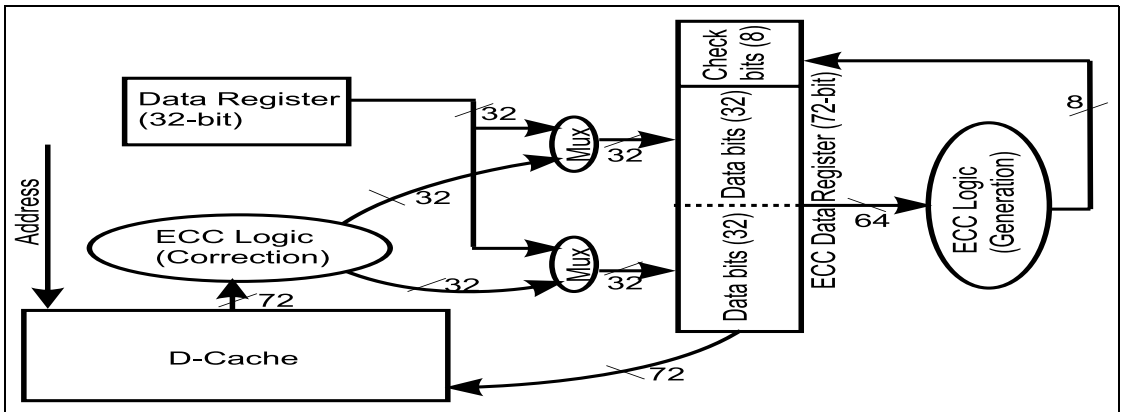

FIGURE 2-5. Illustration of L1 Data Cache ECC-Word Generation. ECC-word generation for a sub-ECC-word store is shown.

One method of achieving soft error protection is to implement ECC-coding directly in the L1 data cache, as is done in the Alpha 21264 [55] and the PowerPC RS64-III [13]. The 21264 and PowerPC RS64-III use 64-bit ECC data words. As shown in Section 2.3.3.1, this provides error coverage for relatively low space overhead of approximately 12%. As also outlined in that section, low cost store verification is trivially implementable as part of ECC check bit generation for subword writes.

In order to illustrate the argument made in Section 2.3.3.1, Figure 2-5 shows a datapath with which ECC may be implemented on a sub-ECC-word size store operation in

an Alpha-like system. We use 72-bit ECC words in the figure because the Alpha uses this word size (and also a slightly modified coding scheme with additional desirable error detection properties) [26].

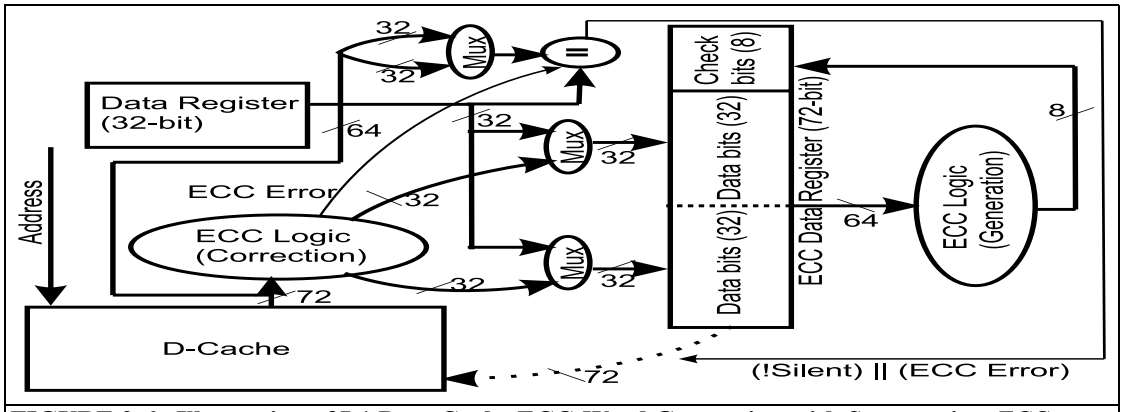

FIGURE 2-6. Illustration of L1 Data Cache ECC-Word Generation with Suppression. ECC-word generation for a sub-ECC-word store with ECC update silent store suppression is shown.

Implementation will be slightly different to handle smaller bit width stores, but for ease of illustration, only a 32-bit store is shown. We see the four major operations as discussed in Section 2.3.3.1: read the original quadword from the data cache, merge the store data into the input side of the ECC Data Register, generate ECC check bits, and store the quadword and ECC bits. Note that if ECC-word generation takes multiple cycles (as one might expect for essentially a read-modify-write sequence), we must maintain atomicity of the sequence either through design of the write buffer feeding the ECC logic, or in the logic itself. We have ignored this detail to simplify the diagram.

In Figure 2-6, we show the implementation of update silent store suppression in the same ECC logic structure as shown in Figure 2-5. We can see that the changes to the datapath are relatively simple; the addition of an extra multiplexor and a comparator. Figure 2-6 also illustrates that we cannot perform update silent store suppression if an ECC error is encountered on the read of the data value from memory. This is because the

corrected value is obtained from the ECC correction logic and therefore must be written back to the memory system. The logic implements the same four steps as described previously. However, the store merge, ECC check bit generation, and new ECC-word store operations may be aborted if it is determined that the store is update silent and there is no ECC error. The abort operation can be as simple as not re-acquiring the cache port for the write of the (update silent) ECC-word from the ECC Data Register.

The most important aspect of Figure 2-6 is when the update silent store comparison can be performed. From the datapath shown, we can see that the comparison can be performed in parallel with the ECC correction logic and check bit generation. In general, ECC correction and generation logic consists of trees of exclusive-or gates [97] which have delay on the same order as the 32-bit comparison for suppression. Therefore, low-cost update silent store suppression for sub-ECC-word stores can be implemented in an ECC-protected L1 data cache for simply the cost of a few extra gates which should not increase the ECC logic's critical path.

2.3.3.3 Write-Through L1 Cache with ECC L2

Implementing ECC protection directly is not the only way to combat soft errors in the L1 data cache. In fact, adding ECC protection to the L1 directly can contribute negatively to cycle time because the ECC correction logic is now added to the critical path on load operations to assure usage of corrected values from the cache. Speculation can be used in order to move the ECC check/correction logic off the critical path by assuming that all load values are correct and recovering if the ECC logic reports an error. Of course, this adds control complexity to trigger the recovery [26].

An alternative is to use an L1 cache with simple parity protection and a write-

through policy backed up with an ECC L2 cache. The L1 parity protection has a few advantages when compared with ECC in the L1. First, parity can easily be kept on a byte basis with the same overhead as the 72-bit ECC-word as in the 21264 (in both cases the overhead is approximately 12%.) With byte parity in the L1, there are no merging issues with store operations because the smallest atom for memory operations is a byte—therefore stores into the L1 do not require a read-modify-write sequence. The parity for each byte can be calculated very early in the pipeline when the store value is known and can simply be written into the cache. The single bit of parity for each byte provides single error detection on the byte level, as opposed to double error detection over 64 data bits as provided by 64-bit SEC-DED. If an error is detected in the L1 data cache via parity, the correct value is fetched from the ECC L2 cache.

A major caveat of this approach is the additional bus traffic generated by implementing a store through L1 cache [52]. This traffic can be reduced with techniques like aggressive write combining and other buffering techniques, but special care must be taken to handle the extra L1 to L2 bandwidth requirements. Weaker consistency models allow greater freedom for store combining than stricter models.

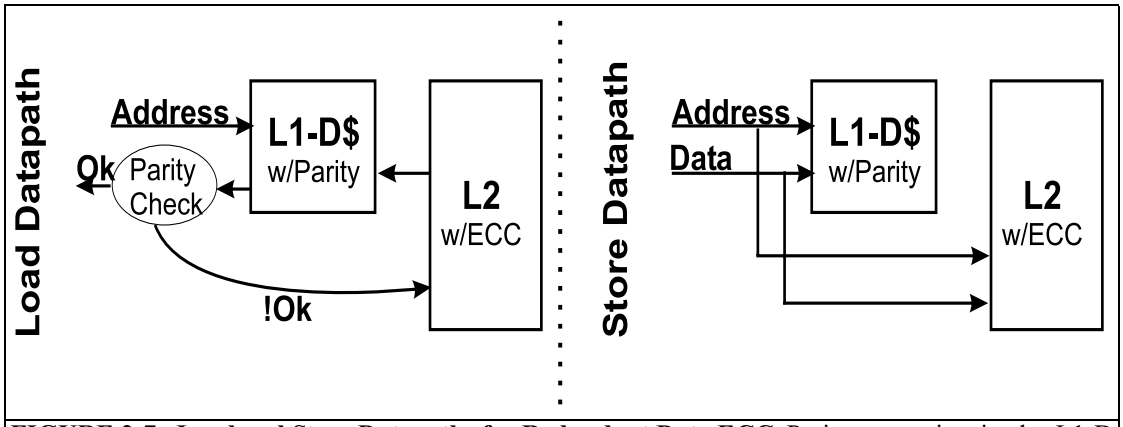

FIGURE 2-7. Load and Store Datapaths for Redundant Data ECC. Parity protection in the L1-D cache with write-through and ECC in the L2 enables ECC in the L1-D cache through data redundancy.

In the case of a store-through L1 cache, we can imagine that update silent store suppression might achieve a noticeable performance benefit by eliminating store-through traffic across the L1-L2 interface because it eliminates writes. We can use the methods explained previously (read port stealing and LSQ locality) to perform low-cost store suppression in such a hierarchy. Performance results for such a configuration are given in Section 2.4.2. It is depicted graphically in Figure 2-7.

2.3.3.4 Duplication of L1 Data Cache

We can also obtain single error detection and correction capability in the L1 cache by duplicating it and protecting both copies with parity. If a parity error is encountered on the read of any byte, the correct byte is fetched from the other copy of the cache to recover from the error. This scheme avoids a read-modify-write sequence for sub-word stores. It also provides effectively double the read-port bandwidth into the L1 data cache because each copy of the data cache can be accessed with loads to arbitrary addresses.

However, this scheme is not without its flaws. First, this scheme has high overhead of 100% compared to a cache with only parity. Second, this scheme does not allow easy scaling of store bandwidth because both copies must be consistent, requiring stores to write both copies.

Reduced cost update silent store suppression may still provide performance benefit in this cache structure because it is biased toward more read ports than write ports. We have argued previously that update silent store suppression is likely beneficial in any architecture which has a higher relative microarchitectural cost for stores than loads. We present simulation results for such an architecture in Section 2.4.2. It is depicted graphically in Figure 2-8.

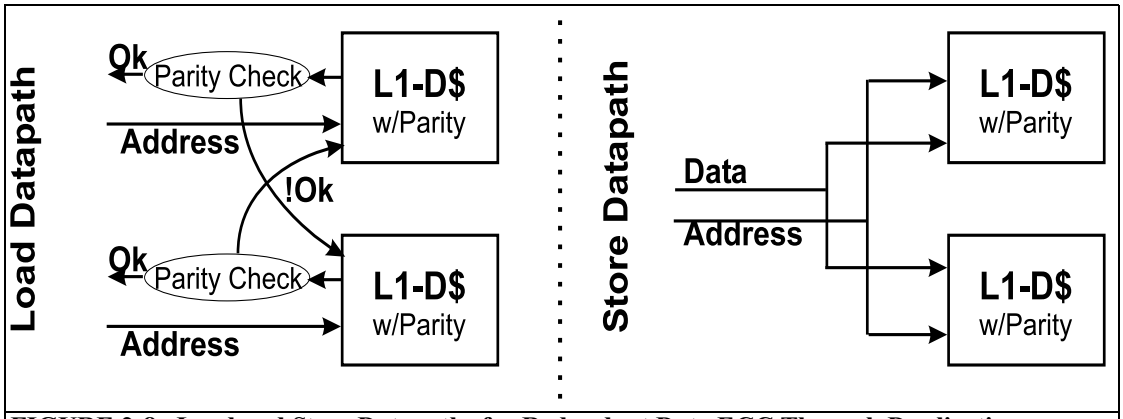

FIGURE 2-8. Load and Store Datapaths for Redundant Data ECC Through Duplication.

2.3.4 Simulation Parameters and Machine Model

We have indicated throughout the previous sections that many opportunities exist to improve performance by exploiting update silent stores in different memory hierarchies. In order to illustrate the performance potential, we present data from each major class of memory hierarchy (writeback L1-L2, and write-through L1-L2). Both hierarchies are prevalent in historical as well as present commercial processor designs, motivating presentation of both. Current examples of write-back include Pentium Pro [25] and Alpha 21264 [26]; write-through includes Pentium4 [30], UltraSparc III [63], and Power4 [105].

Table 2-2: Base Simulator Configurations for Write-Back and Write-Through Machines.

Attribute	Value (Write-Back)	Value (Write-Through)
Fetch/Decode/Issue/Commit	8/8/8/8	8/8/8/8
Pipeline Depth	5	5
BTB/RAS	512 sets, 4-way/8-entry	512 sets, 4-way/8-entry
Branch Predictor	64K G-Share, 16-bit history	64K G-Share, 16-bit history
RUU/LSQ	64-entry/32-entry	64-entry/32-entry
Integer Resources	6 ALUs (1), 2 mul/div (3/12);	6 ALUs (1), 2 mul/div (3/12);
Memory Resources	Multiple*	2 load/store
Floating Point Resources	2 add/sub (4), 2 mul/div (4)	2 add/sub (4), 2 mul/div (4)
L1-I Cache	64KB, 2-way, 64B lines (2)	64KB, 2-way, 64B lines (2)
L1-D Cache	64KB, 4-way, 32B lines (2)	64KB, 4-way, 32B lines (2)
L2 Cache (unified)	512KB, 8-way, 64B lines (8)	512KB, 8-way, 64B lines (8)
Memory/TLB	50 cycles/256 entry I/D	50 cycles/256 entry I/D
L1-L2 Bandwidth	16B/cycle	Multiple*
Write Buffers	2 (8 byte wide)	2 (Multiple*)

The machine configuration used for both hierarchies is indicated in Table 2-2. All binaries are compiled for the SimpleScalar [14] target architecture using gcc at -O3. We evaluate the SPEC-95 and the subset of the SPEC-2000 integer benchmarks set up for our simulation environment for reduced inputs (SPEC-95) and test inputs (SPEC-2000). The subtests presented from SPEC-2000 are chosen based on availability of binaries and compatibility with SimpleScalar 3.0b PISA, our simulation environment for uniprocessor studies. In Section 1.3 we indicated that update silence pervades program execution (across multiple benchmarks, ISAs, and operating environments), indicating that results from this simulation environment are indicative of the opportunity which will be available in other environments as well.

The simulation environment is largely based on SimpleScalar 3.0b [14], with substantial modification to improve modeling of contention and bandwidth within the memory system. For example, we have added the ability to model finite store buffers, and writeback buffers. We have also added an accurate L1-L2 interface model which allows varying bandwidth over the interface, writeback/writethrough modeling, write-combining, etc. We prioritize demand traffic, i.e. load/store misses, over writebacks/write-throughs for all interfaces. Since this thesis is primarily focused on memory system research, such modifications are required. Furthermore, although we use the same core parameters throughout multiple experiments, to provide additional insight, we vary different memory system parameters throughout the results presented. Parameters that should be considered variable are indicated with a (*) in Table 2-2. Furthermore, we assume a weakly ordered memory system in our uniprocessor studies (we explore consistency issues in detail in Chapter 4).

When presenting data on verified update silent stores, we count an update silent store as verified if it begins verification before reaching the head of the RUU. However, we have observed empirically that waiting for all stores to verify before committing them does not yield the best performance; store misses awaiting verification can stall retirement within the RUU, limiting exposed ILP, whereas the base machine would retire these stores into a write-buffer. Therefore, we attempt to verify all stores, but if a store reaches the head of the RUU before verification has completed, it is assumed non-update silent and enters a write-buffer. In this case, the store is performed as normal, regardless of whether it is update silent.[2] Furthermore, we assume that verified update silent stores can release their LSQ entry early, which implies a collapsing LSQ structure. Furthermore, the standard load/store disambiguation policy from SimpleScalar 3.0b is assumed and does not allow loads to issue around update silent stores with unknown addresses as has been proposed and evaluated by Yoaz et al. [111]. Therefore, no benefit is realized from accelerating true memory dependences in this fashion, which would likely improve the results presented here.

2.4 Exploiting Update Silence for Performance Benefit

We show results for two types of memory hierarchies in this section, a writeback L1-L2 cache hierarchy and a write-through L1-L2 cache hierarchy, since both design styles are common in industry. Each hierarchy has different trade-offs at both the core and system-level, which we discuss in the respective sections briefly. For additional insight, the reader is encouraged to refer to Hennessy and Patterson [46] and Culler and Singh

2. It may be possible to suppress such update silent stores in the write buffer or at some other level of the memory hierarchy. Doing so may require transporting the store value throughout levels in the memory hierarchy, a change over traditional store-in (writeback) hierarchies. We discuss write buffer suppression in Section 2.4.1.

[31]. Because both designs are common throughout industry, this thesis does not advocate a particular one, but rather evaluates update silent stores in both hierarchies.

2.4.1 Writeback Hierarchies

Eliminating update silent stores in a writeback hierarchy may achieve reduced pressure on cache write-ports and write-buffering structures throughout the hierarchy, as discussed in Section 2.1. Furthermore, update silent store suppression converts update silent stores into load operations. Therefore, since suppression implies performing fewer stores, fewer dirty cache lines will exist within the memory hierarchy and writeback traffic will be reduced within the memory system.

2.4.1.1 Exploiting Update Silent Stores to Reduce Writebacks

We show update silent store suppression‘s ability to eliminate writebacks between the L1-D cache and the L2 for our simulated machine (with 2 general memory ports) in Table 2-3. The columns indicate the writeback rate in the baseline execution and write-back reductions observed by suppressing L1-D cache hits, L1+L2 cache hits, and all memory references when all stores are verified before leaving the core. We explore verification to varying levels within the hierarchy because verification to higher levels incurs additional latency, potentially impacting processor core performance. As discussed in Section 2.3.4, for the performance results presented all stores need not be verified before leaving the core to obtain best possible IPC. Observed writeback reduction for the configuration used in performance simulations is presented in the final column.

We see from Table 2-3 that suppression can yield a significant reduction in write-backs if all stores are verified before they are allowed to leave the core (L1+L2+Mem column). We see a range in reduction from 0.1% up to 91.8%, with an average reduction of

Table 2-3: Writeback Reduction Between L1-D Cache and L2. The columns indicate the writeback rate, writebacks eliminated by suppressing L1, L1+L2, and L1+L2+Mem store hits when waiting for all store verifies to complete. The final column indicates writeback reduction in the simulated machine configuration (which does not wait for all store verifies to complete).

Benchmark	Baseline WBs /1K Instructions	L1 Hit Suppress WB Reduction	L1+L2 Hit Suppress WB Reduction	L1+L2+Mem Hit Suppress WB Reduction	Performance Simulation Configuration
go	0.40	1.3%	9.8%	13.5%	4.2%
m88ksim	0.27	1.3%	2.1%	57.3%	27.4%
gcc	0.77	7.8%	10.3%	14.3%	7.0%
compress	10	4.7%	57.9%	60.0%	46.5%
li	0.02	1.2%	1.2%	39.7%	19.0%
ijpeg	0.68	0.0%	2.4%	12.9%	6.2%
perl	0.24	6.8%	8.4%	12.5%	9.7%
vortex	5.8	0.7%	17.8%	81.0%	33.2%
gzip	5.8	0.0%	0.3%	0.3%	0.3%
vpr	0.31	0.1%	0.1%	0.1%	0.1%
mcf	22	8.4%	13.7%	91.8%	42.6%
parser	4.8	5.2%	34.3%	43.0%	5.6%
bzip	1.8	0.0%	1.2%	3.1%	1.2%

33%. Furthermore, we note that in most cases store misses to memory must be verified to capture most of the possible benefit. This indicates that many writebacks eliminated through update silent store suppression are in fact due to lines allocated with store misses, as opposed to load misses. The key to understanding this observation is behavior of the dirty bit; lines allocated with load misses will not set this bit until a non-update silent store occurs to the cache line in the L1-D cache (since all L1-D hits are verified across configurations), however unverified store misses will always set this bit regardless of update silence. Since removable writebacks increase substantially when verifying stores which hit higher levels of the hierarchy, this implies verifying the store misses is reducing observed writebacks.

In the configuration used for performance simulation (where store verification need not complete if the store reaches the head of the RUU and the store simply retires into a write buffer as if it is non-update silent), we generally observe writeback reductions

between those where we verify L1+L2 hits and L1+L2+Memory hits. This indicates that such a policy is still capturing significant potential for writeback reduction without sacrificing IPC. Furthermore, note that stores could also be suppressed within the write buffer itself to still achieve the best possible writeback reduction, at the cost of slightly higher write buffer occupancy for non-update silent stores. However, since a store will occupy the write buffer until its data can be delivered into the memory system anyway, the additional occupancy penalty should be extremely low in the case of store misses.[3] Furthermore, depending on the specific design of the memory hierarchy, it may be possible to verify store misses at other levels in the memory hierarchy than the L1, further mitigating additional write buffer occupancy.

2.4.1.2 Suppressing Critical Update Silent Stores for Maximal Writeback Reduction

We have mentioned many possible benefits of suppressing update silent stores in uniprocessors throughout the previous sections, but so far we have only indicated update silent store suppression's ability to reduce writebacks. If we are solely interested in eliminating writebacks, not all update silent stores need be suppressed in order to obtain the maximum possible writeback reduction. Furthermore, naively suppressing can have an impact on core performance as well as system performance when we consider coherence state transitions which take place throughout the memory hierarchy as a function of the memory accesses performed by the core.

In order to formalize these statements, we define the *lifetime* of a cache line as the time between each allocation and replacement of the cache line from a given level in the

3. Some care must be taken in design to minimize additional coherence permission transitions throughout the memory hierarchy for this statement to hold true. We will discuss this further in the context of multiprocessor systems exploiting store update silence (Chapter 4) and also in the context of Critical Update Silent Stores (Section 2.4.1.2).

memory hierarchy, as was previously done in work by Wood et al. [108]. If we are solely interested in eliminating writebacks through update silent store suppression, it is only necessary for us to suppress the *critical update silent stores*, defined as follows:

Definition. A critical update silent store *is a specific dynamic silent store that, if not suppressed, will cause a cache line to be marked as dirty and hence cause a writeback.*

This definition applies for each distinct cache line lifetime. In order to see how the number of critical update silent stores is strictly less than or equal to the total number of update silent stores, consider the following: Each cache line lifetime may have zero to *n* critical update silent stores. Trivially, if there are no stores to the line, there are no critical update silent stores either. Similarly, if there is even one non-update silent store, there are no critical update silent stores (because the single non-update silent store will mark the line as dirty). However, if there are one or more update silent stores to the line and no non-update silent stores, the former set of update silent stores is defined as critical since failing to suppress any of them would result in a writeback.[4]

Suppressing non-critical update silent stores yields no reduction in writebacks because the cache line will be marked as dirty by at least one non-update silent store during the cache line lifetime, by definition. Furthermore, suppressing non-critical update silent store misses can actually degrade performance because we incur the load and compare overhead of a store verify without any compensating reduction in writebacks. This problem is worse in multiprocessors. A noncritical update silent store is replaced with a store verify (read), but a subsequent non-update silent store to the cache line may require the line be upgraded from a shared to modified state (necessitating an upgrade transaction

4. We discuss temporal silence's relation to writebacks and critical silence in Chapter 5.

at the system level). If the non-critical update silent store was not suppressed, the line would have been brought into the cache with a read-with-intent-to-modify (or read-exclusive) transaction, thus eliminating the later upgrade.

We discuss this issue extensively and provide characterizing data in the context of multiprocessor systems in Section 4.4. We introduce the concept here because it is straight-forward to understand in the case of writebacks and this is how we described the phenomenon originally [66] [10]. Furthermore, this observation is relevant in uniprocessors as well, as we describe currently.

In most cases, the additional upgrade transaction will not exist in systems which implement the exclusive or "E" state as known from the MESI protocol [31] because a uniprocessor will obtain most read data in Exclusive state. However, in systems which do not implement the full set of MESI states at every level in the memory hierarchy, e.g. Power4 [105], the notion of criticality still applies, even in uniprocessors. It is entirely possible to implement only MSI at the L1 cache (to maintain circuit speed of the tags, reduce area and complexity, etc.) and only the full MESI state at the coherence point, e.g., the L2 in the examples given. In such a scenario, the non-critical update silent store suppression upgrade can still take place between the L1 and L2. A simple solution to avoid this caveat is to mirror the entire set of stable coherence states at the L1. We detail other possible solutions in Section 4.4.

2.4.1.3 Performance of Simple Suppression Techniques

In Figure 2-9 and Figure 2-10, we show the performance of three different techniques of update silent store suppression described throughout previous sections. The bars, from left to right, shown performance for the baseline case (no update silent store

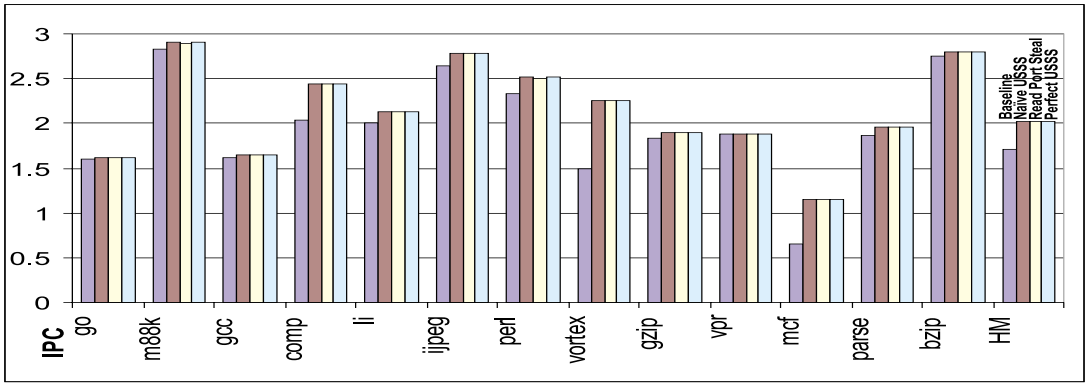

FIGURE 2-9. Performance Comparison of Update Silent Store Suppression Techniques. The bars indicate the performance of the baseline machine, naive update silent store suppression, read port stealing update silent store suppression, and perfect update silent store suppression for a writeback cache hierarchy with 4 load ports and 1 store port.

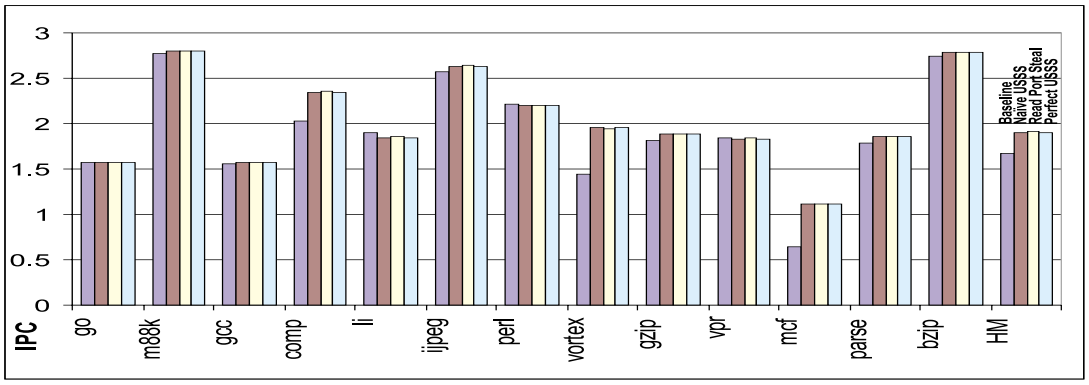

FIGURE 2-10. Performance Comparison of Update Silent Store Suppression Techniques. The bars indicate the performance of the baseline machine, naive update silent store suppression, read port stealing update silent store suppression, and perfect update silent store suppression for a writeback cache hierarchy with 2 memory ports.

suppression), naive update silent store suppression (Section 2.2.1), read port stealing (Section 2.3.1), and perfect update silent store suppression. Perfect suppression utilizes an oracle predictor of update silence and only suppresses the update silent stores—other stores have no verification action carried out during their execution. Therefore, the perfect case eliminates all overheads associated with store verification for non-update silent stores.

Figure 2-9 shows results for a machine configuration with four memory read ports and a single memory write port. As explained previously, such a configuration may be desirable for many reasons; this is a two-wide version of the configuration used in Power4

[105], illustrated in Figure 2-8. Given this memory system design, we expect loads to be substantially less expensive than stores, thus providing potential to benefit from update silent store suppression. Indeed, the figure shows a harmonic mean speedup of 17% for naive update silent store suppression over the baseline, with the improvements ranging from 0% in *vpr* to 76% in *mcf*. As described in Section 2.1, the benefit can come primarily from three factors in this machine configuration: reduced write port/buffer contention, reduced LSQ contention, and reduced writebacks. For the three benchmarks exhibiting greater than 10% IPC improvement (*compress*, *vortex*, and *mcf*), we find that *compress* benefits from reduced write buffer and LSQ contention and *vortex* from reduced LSQ contention. *Mcf* is an interesting case, as it achieves most of its performance benefit from a second-order prefetching effect; suppressing update silent stores effectively issues prefetches for stores at the issue stage instead of initiating a store's access to the memory hierarchy at commit. When exclusive prefetching at issue for store operations is enabled, the improvement in *mcf* is a more modest 6%.

In comparing the three different suppression scenarios, we observe little practical difference for this machine configuration; all results are within 0.1% of one another in terms of IPC. This occurs because store verification interferes little with either non-update silent store commit or load issue, thus negative interference is low. Furthermore, we re-emphasize that the percentage of update silent stores is substantial in general (about 40% on average) implying suppression is very often beneficial.

In order to determine the impact of update silent store suppression when the potential for negative interference for store verification is greater, we also present results for a machine with two memory ports (can be used for either loads or stores, no address restric-

tions) in Figure 2-10. We see similar results, qualitatively, to Figure 2-9. In this case, the harmonic mean speedup of update silent store suppression is 13%, smaller due to negative interference from store verification and also because each dynamic store operation is less expensive when compared with a load. Therefore, eliminating stores provides less memory port contention benefit.

We also observe that read port stealing is the most effective suppression mechanism in this scenario, improving over perfect suppression by 0.3% in harmonic mean. Furthermore, with this machine configuration, *li* actually shows a slowdown due to update silent store suppression of 2.5%, due to its low fraction of update silent stores (20%, as shown in Table 1-1). However, the read port stealing mechanism reduces the slowdown to less than 1.0%, indicating this mechanism is an effective way to reduce negative interference when relatively few dynamic stores are update silent.

Now that we have determined read port stealing is the most effective mechanism of update silent store suppression for writeback hierarchies to minimize negative interference, we examine performance sensitivity with respect to another key memory system parameter: the number of write buffers. As we have stated, eliminating write buffer contention is a potential benefit of update silent store suppression. We explore sensitivity to the number of write buffers implemented outside the processor core in Figure 2-11.

We observe that performance increases for both the baseline machine and one implementing read port stealing update silent store suppression given additional write buffers outside the RUU. More importantly, we observe that the substantial speedups observed in *compress*, *vortex*, and *mcf* previously are reduced substantially. Harmonic mean IPC improvement is reduced from 17% with two write buffers (Figure 2-10) to 4%

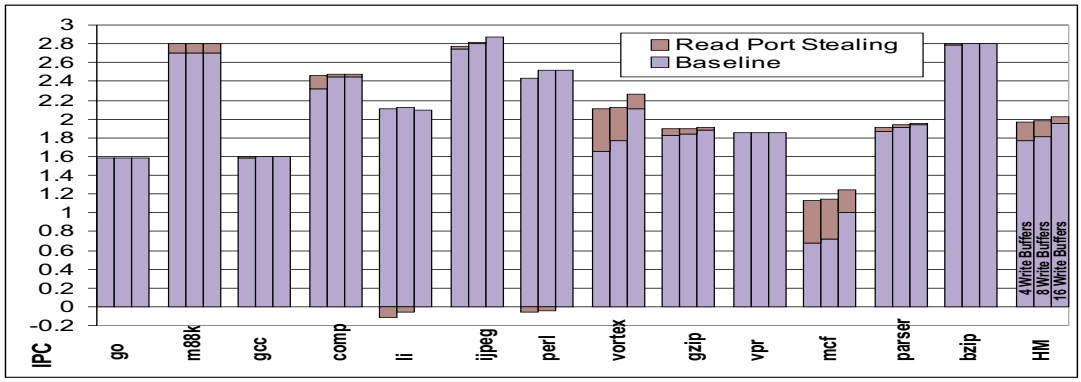

FIGURE 2-11. Performance of Update Silent Store Suppression with Varying Write Buffers. The stacked bars indicate the performance of the baseline machine and read port stealing update silent store suppression for increasing numbers of write buffers (4, 8, and 16) for a writeback memory hierarchy with 2 memory ports.

with 16 write buffers. This result is not unexpected. As the baseline machine's store-handling bandwidth improves, we expect less benefit from suppression. However, performance improvement can still be obtained, even with a well-resourced memory hierarchy.

We have indicated throughout this section that a majority of the performance improvement from update silent store suppression comes from reducing contention for store-handling structures within the processor core and memory system ports. However, in Table 2-3 we indicated that writebacks can be reduced substantially as well by suppressing update silent stores.

Writeback hierarchies can achieve good write performance because of the natural write combining that occurs to lines modified within the L1-D cache before the dirtied lines are written back to other levels in the memory hierarchy. Therefore, most of the performance benefit in writeback hierarchies from update silent store suppression comes from reduced write port contention, write buffer contention, and removed writebacks.[5] For

5. In some cases, e.g. *mcf,* a secondary prefetching effect can be considerable, as store verification loads prefetch store data into the L1-D cache before a store attempts to commit, reducing write buffer occupancy for the store substantially, as opposed to simply trying to cover the entire store's latency within the write buffer.

these workloads and machine configurations, we see very little impact (less than 0.1%) on IPC due to writeback handling between the L1-D and L2 caches as long as there is at least one writeback buffer. Performance improvement from eliminating writebacks may be more substantial in different classes of workloads where writebacks contribute a substantial portion of memory traffic, as described by Lee et al. [65]. These results (insensitivity of these workloads to writeback traffic) are corroborated in that work.

Finally, we point out that results with a performance model which exploits ECC update silent store suppression (Section 2.3.3) in addition to read port stealing are quantitatively similar, and are omitted for the sake of brevity. However, performing read port stealing to suppress most update silent stores in the core and ECC update silent store suppression to eliminate remaining update silent stores can achieve the core benefits of read port stealing and also eliminate all writebacks presented for the L1+L2+Mem column of Table 2-3.

2.4.2 Write-Through Hierarchies

In write-through hierarchies, the natural ability of the L1-D cache to combine references is purposely eliminated. There are many reasons architecturally why this might be desirable: to avoid explicit ECC coding protection of the L1-D cache (as explained in Section 2.3.3), to allow integer and floating point datapaths to be physically located away from one another (as is done in Pentium4 [30] and Itanium2 [80]) while still providing a coherent memory image for the floating point core, and to ease the coherence requirements on the L1-D cache in multiprocessor systems by providing an up-to-date copy of each memory location at the L2 level. However, such an architecture obviously has a much higher cost per store operation than a writeback hierarchy because of the combining

opportunity lost. Therefore, most write-through hierarchies provide aggressive, combining write-buffers to reduce bandwidth required on the L1-L2 interface to handle write-through traffic[6].

Since write-through hierarchies create additional costs for store operations when compared to writeback hierarchies, we evaluate our most aggressive store verification/suppression mechanisms in this context. We expect that these techniques might also provide benefit in writeback hierarchies, but do not rigorously evaluate them in that context for the sake of brevity.

2.4.2.1 Read Port Stealing Performance Benefits

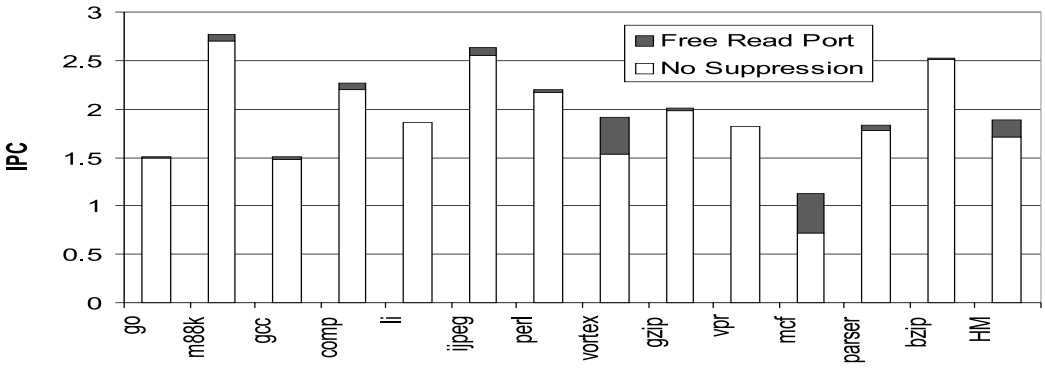

FIGURE 2-12. Performance Improvement of Read Port Stealing. performance of read port stealing update silent store suppression is compared against baseline performance without update silent store suppression.

Figure 2-12 shows the performance improvement of read port stealing over the baseline performance with no suppression. We see improvements ranging from a low of 0% in *li* and *vpr* to a high of 56% in *mcf*. The harmonic mean across all benchmarks shows a 10.3% improvement.

It is worthwhile to note that we do not see a performance decrease in any bench-

6. Memory consistency considerations for when write-buffering/combining is legal are not considered in these results. We assume a weakly ordered memory system in our uniprocessor results.

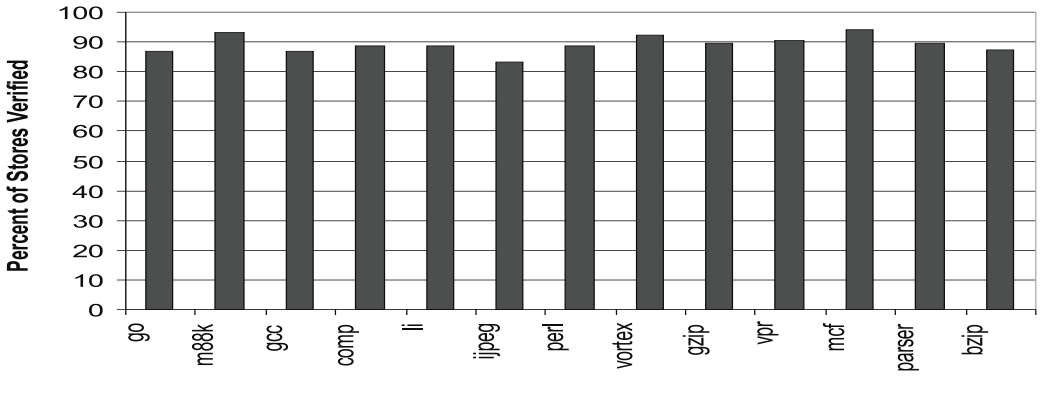

FIGURE 2-13. Dynamic Stores Verified Using Only Available Cache Read Ports.

mark. This occurs because we are only using cache read ports available after all other ready loads and stores have had a chance to issue/commit. The performance benefit comes primarily from three factors: a) a reduction of bandwidth required between L1 and L2 caches by eliminating store traffic on the interface, b) reduced pressure on write buffers, and c) reduced contention for entries in the LSQ. In some cases (especially noteworthy in *mcf*) a secondary prefetching effect from store-verifies contributes substantial speedup. Store verifies essentially issue prefetches for stores at issue, allowing for overlap of store latency within the RUU before stores retire. Without this prefetching effect, multiple store-misses easily overwhelm the limited store buffers in the machine, causing the RUU to stall. We verified this result by adding exclusive prefetching at store issue to our base model. Compared against this baseline, *mcf* achieves a non-trivial, but dramatically smaller, performance improvement of 9%.

It is also interesting to note how few store suppression opportunities we miss by only using available cache read ports as opposed to trying to suppress all stores. In Figure 2-13 we show the percentage of store operations we are able to store verify for free using read port stealing.

We can see that in all cases, we are able to verify over 83% of store operations

using available cache read ports with an average of 89%. This indicates that we are achieving almost all available benefit from suppression that uses the standard store verify, but without impacting performance of critical load and store operations.

2.4.2.2 Load Store Queue Suppression Performance Benefit

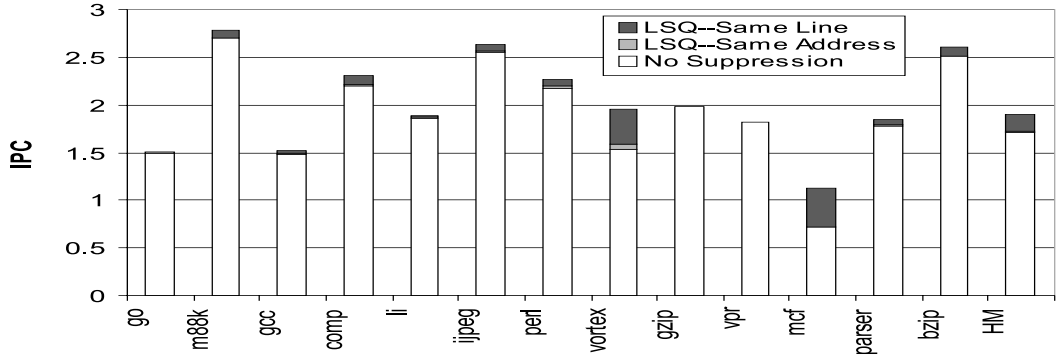

FIGURE 2-14. Performance of LSQ Suppression. The stacked bars indicate the performance of the baseline system (without suppression), same address (temporal) LSQ suppression, and same cache line (spatial) LSQ suppression, respectively.

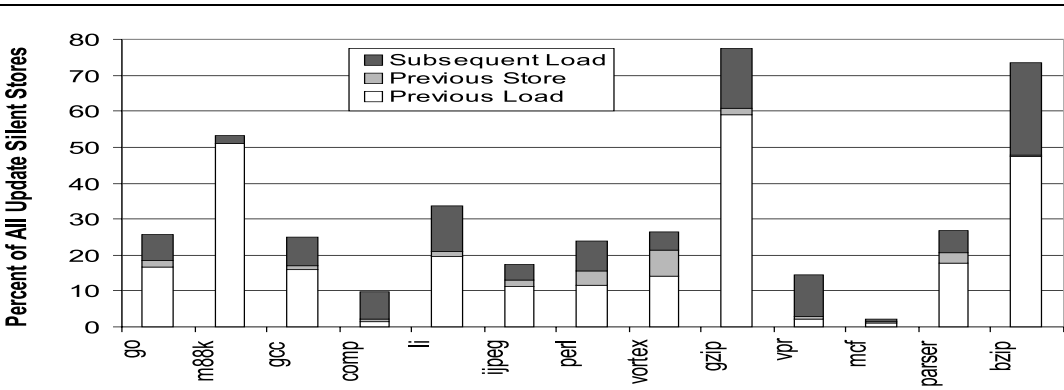

FIGURE 2-15. Temporal LSQ Suppression Through WAR, WAW, and RAW Dependences.

In Figure 2-14, we show the performance improvement of temporal and spatial LSQ suppression over the baseline performance with no suppression (as discussed in Section 2.3.2.1 and Section 2.3.2.2, respectively). The stacked bars show the contribution of each mechanism to overall performance. For temporal LSQ suppression, we see improvements in IPC ranging from a low of 0% in *gzip* and *mcf* to a high of 3% in *vortex* with overall performance improved by 0.6% as indicated by the harmonic mean over all bench-

marks. When we add spatial LSQ suppression, we see total improvements over the baseline from a low of 0% in *gzip* to a high of 56% in *mcf* with the harmonic mean improving by 11.3%.

When examining temporal suppression, it is interesting to note that most of the stores are suppressed by preceding or subsequent load operations (the RAW and WAR dependences discussed in Section 2.3.2.1), as opposed to previous store operations (WAW dependences), as illustrated in Figure 2-15. In most benchmarks (except *compress*, *ijpeg*, *vpr*, and *mcf*), temporal LSQ suppression captures over 25% of all update silent stores within the dynamic program execution. Some possible explanations for this are provided by Bell et al. [10] and in Moshovos's thesis [85], and could include program model considerations like stack frame usage. In the results presented in Figure 2-15, each dynamic update silent store is counted at most once (it is present in only one section of the stacked bars), with the following priority counting on multiple aliases: previous load (WAR), previous store (WAW), subsequent load (RAW).

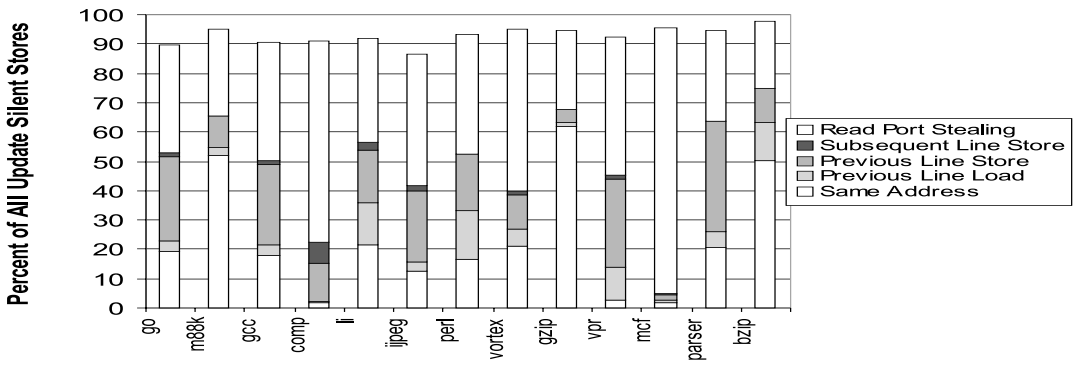

FIGURE 2-16. Sources of LSQ Store Verify Data. The percentage of update silent stores verified using data provided by same address aliases (temporal locality), previous load to line, previous store to line, subsequent store to line (spatial locality), and read port stealing is indicated.

In the case of spatial LSQ suppression, the same statement regarding counting of suppressible stores holds (a dynamic update silent store is only counted once). However,

the priority of counting changes slightly due to simulator implementation issues. In this case, the counting precedence is: WAR, WAW, cache line previous load, cache line previous store, RAW, cache line subsequent load, and read port stealing. We show the results of this method of counting in Figure 2-16 (results from all same address suppression methods are combined in the Same Address bar for readability and the subsequent line load section is removed because it did not contribute meaningfully). Note that the total percentage of update silent stores captured by this mechanism is greater than the results presented in Figure 2-13 (read port stealing) because stores verified from data in the LSQ cache do not consume a cache port. Therefore, a port tends to be free more often for additional read port stealing store verifies.[7]

We see that the percentage of same address store verifies decreases over Figure 2-15, mainly due to counting precedence. Also, substantial previous line store verifies are observed, indicating that the LSQ cache proposed in Section 2.3.2.2 is useful. These results also indicate, due to the small fraction of subsequent line verifies, that verification from a line allocated by a subsequent access to previous stores in the LSQ is unnecessary for suppression purposes, potentially saving some complexity in the LSQ cache.

Finally, we see that in all benchmarks (except *compress* and *mcf*), over 40% of all update silent stores are captured by exploiting locality in the LSQ. Read port stealing for LSQ cache line allocation brings the total percentage of update silent stores captured to over 90% (except for *ijpeg*).

---

7. In machines which rely on speculative scheduling [12] this serialization of lookups in the LSQ cache may not in fact save a cache port, as the port usage will be scheduled before the hit status in the LSQ cache is known. However, serializing the lookup may still be beneficial in terms of power (saving access to the cache arrays), similar to Yang and Gupta [109]. Furthermore, accurate update silence predictors have been developed [57, 111], which may still allow saving a memory port.

In comparing temporal to spatial LSQ suppression, we see only two benchmarks that benefit from temporal suppression (*perl* gains 1.5% and *vortex* 3.3%). It is not until spatial LSQ suppression is applied that we see noticeable improvements in instruction throughput. This occurs because the overall percentage of update silent stores detected by the spatial scheme (including free read port suppression) is much higher.

2.4.2.3 Increasing Write Through Bandwidth with Update Silent Store Suppression

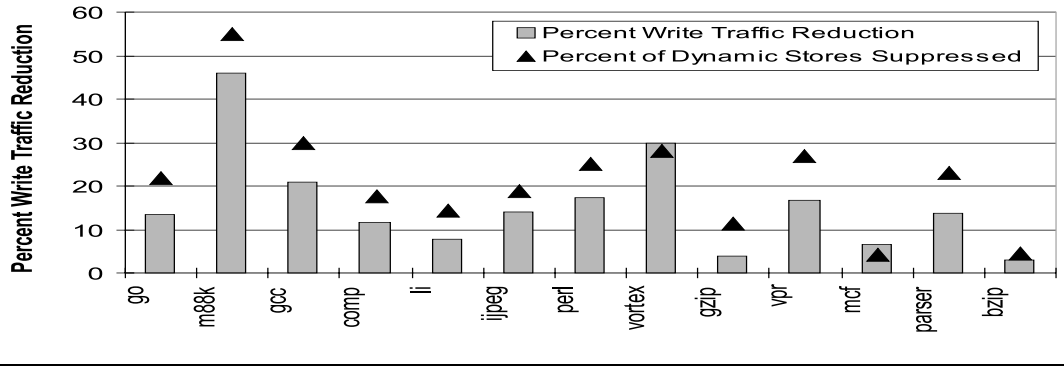

FIGURE 2-17. Percent Reduction of L1 to L2 Traffic with Advanced Suppression Techniques. The bars and bullets (tip of the triangle) indicate the percentage of write through traffic reduction and the percentage of total dynamic stores removed, respectively.

Given that advanced techniques can suppress many update silent stores, it is interesting to examine what kind of trade-offs we can make as an architect with this type of memory system to obtain sufficient throughput between the L1 and L2 caches. We can use the brute force method and implement a fully-pipelined, write-combining, cache-line-width interface between L1 and L2 (as used in all results presented so far) which can induce significant circuit design complexity. Or, we can exploit update silent store suppression to obtain effective throughput over the L1 to L2 interface with less physical throughput. In order to illustrate this, we present Figure 2-17 which shows the store-through traffic reduction over the L1 to L2 interface as well as the percentage of dynamic stores removed by update silent store suppression. We see an average traffic reduction of

15% across all benchmarks and up to 45% in *m88ksim*. Since this interface is wide (32B) and fast (single cycle pipelined), it is reasonable to assume that this traffic reduction would lead to a savings in chip power.

Note that, as we would expect, the percentage of write through traffic reduction closely mirrors the percentage of successfully suppressed stores. In the case of *vortex* and *mcf*, the traffic reduction is slightly greater than the percentage of removed stores, which we determined was due to a second-order effect of increase in write combining efficiency; because suppressed stores do not allocate a write buffer, there are more buffers available for combining non-update silent stores. The percentage of removed stores is lower than the overall percentage of update silent stores (and also the percentages of suppressed stores presented previously) because we do not wait for store verifies to complete before committing stores (explained in Section 2.3.4). In further experiments not detailed here, we found that although traffic was decreased by waiting for stores that hit in the L1 to finish verifying, because commit of some stores is stalled in this case, overall instruction throughput is lower. There is a potential performance vs. power consumption trade-off here that could be exploited in power-aware designs.

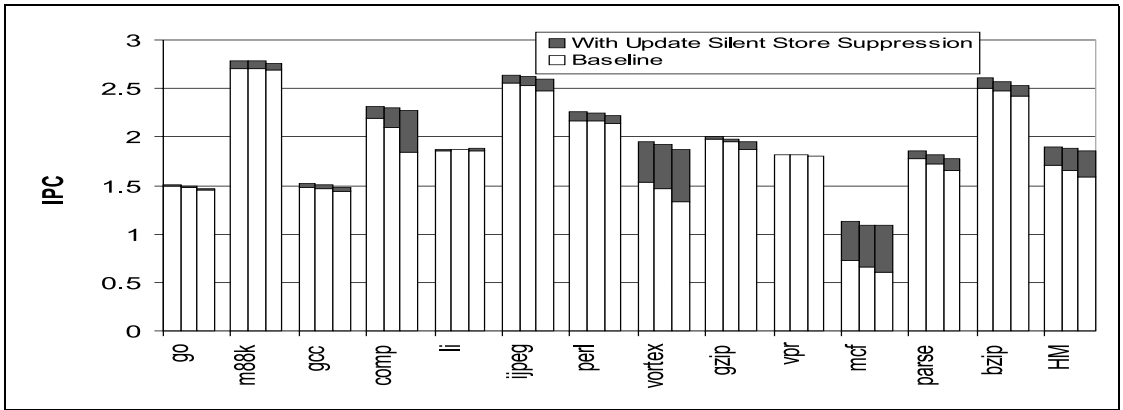

FIGURE 2-18. Performance Comparison for Narrowing L1 to L2 Interfaces. The stacked bars indicate the performance obtained with suppression as compared to baseline performance for 32B, 16B, and 8B wide L1 to L2 interfaces, respectively.

In order to determine how effective this bandwidth reduction is on instruction throughput, we present Figure 2-18, which shows the performance across all benchmarks with varying width interfaces between the L1 and L2, with and without update silent store suppression in its most aggressive form (spatial LSQ suppression with read port stealing). We keep the L1 cache line size at 32B in all simulations, but illustrate the performance of 32B, 16B, and 8B wide interfaces between the L1 and L2. In each case, the physical bandwidth of the L1 to L2 interface is progressively lowered; in the case of 16B and 8B widths more transactions across the interface are required for a cache line transfer (two and four transactions for 16B and 8B, respectively). However, the write combining width is changed to match the physical interface width so that flushing a write buffer takes only a single cycle.

If we compare update silent store suppression with an interface width of 8B to no suppression with an interface width of 32B, we see that the effective bandwidth (as evidenced by IPC) of suppression with the 75% lower physical bandwidth interface is more effective than the higher physical bandwidth interface without suppression (the only exceptions to this are *go* and *gzip;* in these benchmarks, the percentages of update silent stores are low, 27% and 16% respectively, leading us to expect less benefit from suppression). In fact, as evidenced by the harmonic mean, the update silent store suppression low physical bandwidth interface actually provides 9% greater effective bandwidth on average than the fastest physical interface modeled. Therefore, we can potentially trade the implementation of suppression for physical bandwidth. Of course, as also shown, update silent store suppression still provides benefit no matter what physical bandwidth is available. Note that even though the actual reduction in physical bandwidth for the narrower inter-

faces (50% and 75% for 16B and 8B wide interfaces, respectively) is larger than the percent reductions shown in Figure 2-17, suppression also decreases pressure on other hardware structures, such as write buffers, so the performance improvement is not solely due to the reduced L2 bandwidth.

We also observe in Figure 2-18 that the performance degradation from the widest (32B) to the narrowest (8B) interface is lower in the case of update silent store suppression than for the baseline system with no suppression (40% lower according to the harmonic mean). This occurs because suppression is relatively more effective as the write-combining width narrows. With respect only to physical interface bandwidth, combining and suppression are equivalent. We can either save a transaction over the L1 to L2 interface by combining with a previous store or by suppressing the store. However, there is some overlap between the methods—some stores that are suppressed could also have been combined, and vice-versa, as can be seen in Figure 2-17 in the difference between removed dynamic stores and reduced write through traffic. Of course, the combining width directly affects the number of stores that can be combined, but does not directly affect the number of suppressed update silent stores. Therefore, suppression will capture some stores that can no longer be combined (but can still be suppressed at any combining width), so the relative benefit of suppression increases as the combining width decreases along with the width of the L1 to L2 interface.

As in Section 2.4.1.3, we explore the performance sensitivity to available resources within the memory hierarchy, namely the number of write buffers. Intuitively, we expect that as the microarchitecture's ability to handle store bandwidth demand improves, less performance benefit will be obtained with update silent store suppression.

Figure 2-19 shows the performance of the baseline machine and one implementing update silent store suppression for the 8B-wide L1-L2 interface for increased numbers of store buffers (4, 8, and 16).

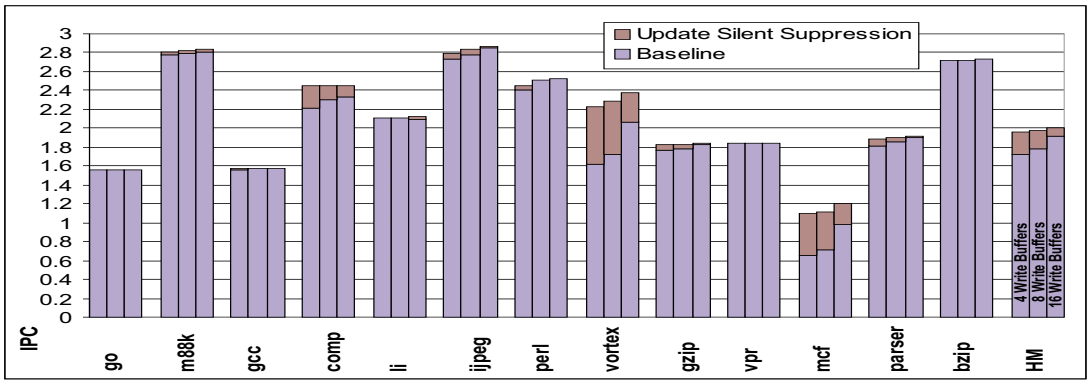

FIGURE 2-19. Performance Sensitivity to Increased Write Buffering. The stacked bars indicate the performance obtained with suppression as compared to baseline performance for an 8B wide L1 to L2 interface, with 4, 8, and 16 write buffers, respectively.

Once again, we see the performance of both the baseline machine and one implementing suppression improves as write buffers are added. As also observed there, the relative performance benefit of suppression decreases as write buffers are added, reducing the harmonic mean performance increase from 16% for two write buffers (Figure 2-18) to 5% for 16 write buffers. Again, this is not unexpected. However, even with a well-resourced memory system performance improvement is still achieved. Detailed explanations for each benchmark showing substantial performance improvement (*compress*, *vortex*, and *mcf*) has been given throughout this, and previous, sections.

2.4.2.4 Discussion of Aggressive Update Silent Store Suppression Mechanisms

Comparing the performance results for the three update silent store suppression methods simulated for our machine model, we see that read port stealing and aggressive LSQ suppression exploiting both temporal and spatial locality in the LSQ provide nearly equivalent performance, with harmonic mean speedups of 10% and 11%, respectively.

This occurs because both methods capture greater than 83% of all update silent stores and close to 90% on average, so both methods are suitable for achieving IPC benefit. However, aggressive LSQ suppression reduces the number of store verifies issued to the memory system by 50%, on average (comparing the read port stealing percentages from Figure 2-13 and Figure 2-16). Therefore, in a machine model with relatively fewer memory ports, aggressive LSQ suppression may have greater benefit because of reduced port contention. Reducing data cache accesses may also reduce overall power consumption.

Temporal LSQ suppression by itself provides only modest speedup in these benchmarks, less than 1%, because of the low percentage of update silent stores captured (31% on average) and the corresponding 9% average reduction in total committed dynamic stores. Therefore, while temporal LSQ suppression has the benefit of never stealing a cache read port, in our machine, solely implementing this mechanism does not seem worthwhile.

Finally, we note that performance benefit from update silent store suppression is strongly correlated to the relative cost of load and store operations within a given memory hierarchy. As discussed at the beginning of Section 2.4, the only fundamental ILP improvement from exploiting update silence within the core is by removing true store to load dependences on update silent stores. Therefore, for core designs with substantial write handling bandwidth at all levels within the memory hierarchy which do not relax true store to load dependences we expect relatively less benefit from suppression. If update silent stores are exploited to remove true store to load dependences, we expect benefit to be proportional to the fraction of such dependences. Other authors have shown a strong variability of such dependences on instruction set architecture (mostly correlated to

the number of architected registers) and also instruction window size [111, 85]; therefore, potential benefit from this optimization will be strongly correlated to these aspects.

As we have also discussed at length throughout Section 2.1 and Section 2.3.3, implementing multiple cache write ports for wide-issue processors or sufficient store bandwidth considering ECC for future deep-submicron technologies may make stores more costly. Therefore, naively assuming store bandwidth can be scaled may be perilous. In any event, we have shown that for many types of memory systems much write traffic can be eliminated; actual performance benefit achieved is of course related to specifics of the memory system design.

2.5 Exploiting Temporal Silence To Improve Core Performance

Experimental results (not presented here for brevity) indicate that there is little benefit to exploiting temporal silence, introduced in Section 1.4, for uniprocessor speedup in ways similar to those presented throughout this chapter. The principle explanations for this include:

- Reducing cache writebacks for most traditional applications does not seem to affect final core performance measurably as long as sufficient buffering is provided (there are some counter-examples for certain streaming workloads, e.g. Lee et al. [65].)
- We have not devised a method to exploit temporal silence at any significant program distance for cache port utilization reduction due to requirements of precise exceptions.
- We have observed that the dynamic window of stores we must examine to capture significant temporal silence opportunities in write-through hierarchies requires rel-

atively large write buffers for single-threaded applications. Enlarging these structures sufficiently to capture temporal silence also improves their ability (without exploiting store value locality) to eliminate write-throughs. Therefore, exploiting temporal silence to further eliminate write-throughs provides little gain over update silent store suppression.[8]

However, efficiently detecting temporal silence in the processor core and on-chip memory system is important for multiprocessor systems, as we will show in Chapter 5. Therefore, we defer discussion of efficient methods of detecting and exploiting temporal silence until that time.

2.6 Related Work

In related work with collaborators, we have characterized update silent stores further by examining the correlation of store update silence to memory region (stack vs. heap), memory hierarchy, i.e. dynamic program distance, past address and value behavior for specific static stores, update silent store value distributions, dynamic execution frequency, fraction of static stores contributing dynamic update silent stores, and compiler optimization level [10]. We have also examined source code (for SPEC-95) attempting to gain insight into why update silent stores exist. We have also indicated that update silent stores add a new dimension to classically "dead" stores, although silence and deadness are orthogonal [19].

The results presented in this joint work indicate that it appears there is no single or simple way to eliminate the vast majority of update silent stores; i.e. update silent stores

8. This is not necessarily true in multithreaded applications where we can exploit temporal silence to eliminate communication misses (see Chapter 5). However, requirements for maintaining memory consistency preclude simple write buffer enlargement from being effective (see Chapter 3) without employing speculation.

are not simply due to a specific programming language artifact or a single common code construct, and do not seem to be easily handled with compiler techniques. Empirical evidence presented in this work (by studying compiler optimization level) indicates that update silence seems to be strongly correlated to program behavior and not compiler artifacts; in fact, as compilers optimize more aggressively the percentage of update silent stores tends to increase (we have also observed this across compilation environments when moving from a slightly outdated version of gcc used to generate SimpleScalar binareis, to a production, heavily optimizing compiler, xlc for AIX, as shown in Table 1-1). Given dynamic runtime information there are a few strong correlations observed for store update silence—e.g., nearly 50% of dynamic update silent stores write the value zero; static stores which write the same value as the last time they were executed are more likely to be update silent than those which write a different value, etc. However, eliminating the vast majority of update silent stores still appears to be non-trivial. These studies also did not examine multi-threaded workloads which provide additional opportunities for exploiting store value locality.

Calder et al. [19] discussed the concept of an update silent store in terms of hoisting loads and dependent instructions above loop bodies. They proposed extending the memory disambiguation buffer (MDB) to not only check for store aliases but also whether the location's value had changed. Molina et al. [82] proposed exploiting redundant stores for eliminating cache writebacks. Yoaz et al. [111] examine exploiting store update silence to advance later program loads around unresolved stores which are highly likely to be update silent thus avoiding store to load forwarding aliases which will not in fact affect a correct value being delivered to the load operation.

Update silent stores have also been explored as a means to eliminate cross-thread dependences in thread-level speculation systems (TLS) [103, 24]. Results in these works showed that a substantial number of cross-thread memory dependence violations signaled solely considering that a location was written can be eliminated by suppressing update silent stores. In related work, others have proposed exploiting update silent stores in Slipstream processors [104] for similar reasons.

We have mentioned at various points throughout this chapter the potential power and energy reductions which may be realized by eliminating update silent stores. However, we are principally focused on performance in this thesis. Yang and Gupta have described a mechanism to remove "redundant load and store" memory operations to decrease cache power [109]. They classify a store as redundant if it is update silent; however the mechanisms devised to take advantage of store update silence differ from our methods of update silent store suppression as they are principally interested in minimizing data cache accesses.

Finally, we discuss many optimizations possible through exploiting both temporal and spatial locality inside the LSQ to enable low-cost suppression of update silent stores. Moshovos has studied this extensively in his thesis, illuminating reliable methods of memory dependence prediction (predicting dynamic memory references to the same location) and also anti-dependence prediction (predicting dynamic memory reference to different locations) [85].

Chapter 3

Multiprocessor Sharing Considering Store Value Locality

In Chapter 1 we introduced update silent stores and temporally silent stores. We further described the basic mechanisms of invalidation-based cache coherence and discussed communication misses in Section 1.1. Communication misses occur in cache coherent systems in order to propagate changes to shared-memory state to other processing elements in the system. Of course, these changes to shared-memory state are a direct consequence of store operations. In this thesis, we are illuminating the fact that substantial store value locality exists; this unearths the potential to eliminate communication misses by exploiting the value locality of store operations. For the rest of this thesis, we focus on this potential application of store value locality. Although we predominantly focus on broadcast protocol, shared-memory multiprocessors, many of the results presented here may be applicable in other multithreaded programming environments and also implicitly threaded environments. We will discuss this related work, as well as opportunities for follow-on research throughout Chapter 4, Chapter 5, and Chapter 6.

3.1 Motivation and Background

There is widespread agreement that communication misses are one of the most pressing performance limiters in shared-memory multiprocessor systems running commercial workloads. For example, both Barroso et al. [9] and Martin et al. [75] report that about one-half of all off-chip memory references are communication misses; i.e. the references are satisfied from dirty lines in remote processor caches. Communication misses are caused by remote writes to shared cache lines; in single-writer or invalidate protocols a write requires all remote copies of a shared line to be invalidated [31]. Subsequent refer-

ences to those remote copies lead to misses that must be satisfied from the writer's cache. Two current trends are likely to exacerbate this problem: systems that incorporate an increasing number of processors will likely lead to an increased probability that a remote write to a shared line will occur, and systems with larger and more aggressive local cache hierarchies that eliminate most capacity and conflict misses, but cannot reduce communication misses.

We discussed the concept of a dynamic cache line lifetime in the context of write-backs in uniprocessors in Section 2.4.1.2; the lifetime begins when a cache line is allocated and ends when the line is replaced (and written back if dirty) from the cache. In the context of multiprocessor systems, an additional event can also cause the lifetime of the cache line to end, namely an invalidation request from another processor in the system. In order to reduce the number of communication misses, we strive to extend the lifetime of a cache line throughout processors in the system. Intuitively, if we can extend the lifetime by either reducing the number of, or judiciously applying, coherence events against cache hierarchies throughout the system, we should be able to improve performance. This is reasonable because we are improving the likelihood that memory references can be serviced with data already resident within the local cache hierarchy, thus avoiding a transfer from memory or the most recent writer's cache (so called "modified interventions" or "dirty misses"). We depict existing methods of extending cache line lifetime related to multiprocessor coherence in Figure 3-1.

Prior work has shown that many communication misses can be avoided by detecting false sharing [35]. As illustrated in Figure 3-1, this approach attempts to extend the lifetime of a shared copy of a cache line by monitoring remote writes more closely, and

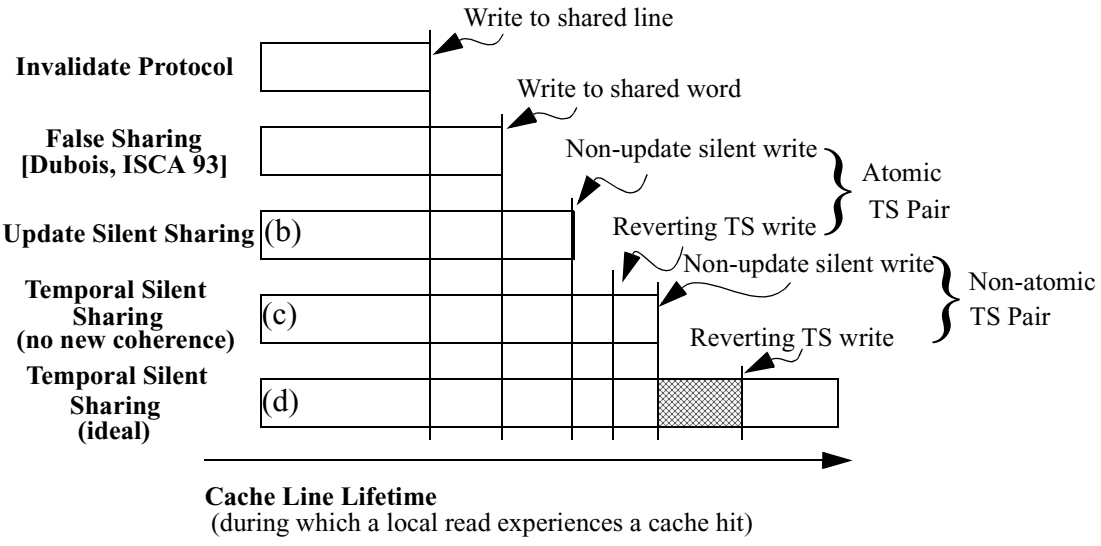

FIGURE 3-1. Extending the Lifetime of a Shared Cache Line. Lifetime is extended by avoiding writes to other parts of the word (false sharing), ignoring update silent writes (update silent sharing), ignoring temporally silent atomic write pairs, and reinstating a line upon reversion due to a temporally silent write pair that is not atomic.

extending the lifetime of the cache line whenever the remote write changes an unaccessed word in the line. Such misses due to cache line granularity (and not intrinsic sharing within multithreaded program execution) were first described by Goodman et al. [43].

Dubois et al. have rigorously classified misses in the context of infinite-cache, invalidation-based, multiprocessor systems [35]. We reintroduce relevant definitions from that work here so that we can naturally extend them to consider the value locality of stores. The relevant definitions are as follows:

Cold Miss: The first miss to a given block by a processor.

Essential Miss: A cold miss is an essential miss. Also, if during the lifetime of a block, the processor accesses (load or store) a value defined by another processor since the last essential miss to that block, it is an essential miss.

Pure True Sharing miss (PTS): An essential miss that is not cold.

Pure False Sharing miss (PFS): A non-essential miss.

Essential misses constitute all misses which bring in a truly shared word either directly, or as a side effect (for example, when a truly shared value is brought in as the non-critical word in a cache refill). Note that the use of the word “value” in the above definition means value in the invalidation sense only, i.e, a store instruction has occurred to that address. It is not implying anything about the data value at that address.

In general, Dubois contributed the insight that merely tracking the address that invalidates a cache block or only comparing the address that causes a miss to previously written locations in that block is not sufficient. To be more precise, we must examine all previous invalidations of a block and the side-effects of loading a cache line (including future accesses to data within the cache block) to be sure that PTS and PFS misses are correctly attributed.

3.2 Update Silent Sharing

As you might expect, studies (detailed in Chapter 4) have determined that update silent stores can be exploited to further extend cache line lifetimes. This can be easily understood when considering the proposed methods for update silent store suppression (Section 2.2) which essentially convert update silent stores into load operations—which do not require exclusive ownership. This is depicted graphically as Figure 3-1(b); the lifetime is extended until a non-update silent store occurs to the cache line.

To rigorously define update silent sharing, we keep the same definitions as Dubois with extensions covering the value locality of stores. Intuitively, we extend the definition of essential miss to exclude those stores which are update silent, i.e, those that do not change the machine state because they are attempting to store the value that was previously available at that location in the system memory hierarchy. Rigorously, we propose

the following, modified, definition of an essential miss (our changes are in *italics*): 61

Essential Miss: A cold miss is an essential miss. Also, if during the lifetime of a block, the processor accesses (load or store) *an address which has had a different data value* defined by another processor since the last essential miss to that block, it is an essential miss.

While the wording of this definition is almost the same as the one proposed by Dubois, we have made a slight change to make clear that we are interested in the data value at a memory location. The other definitions remain accurate with no modification. Note that within the definition of Update Silent Sharing (USS), the possibility for both true sharing and false sharing misses still remain. Therefore, when presenting classification of sharing misses considering USS, miss reductions over the baseline case, i.e. Dubois' classification, are represented as a reduction in overall misses for ease of presentation. The reduction in each type of miss can therefore be determined by directly comparing the reduction in true sharing/false sharing misses between the baseline and USS.

In previously published work related to this thesis, i.e. [10, 68, 66, 69], we had defined this as Update False Sharing. However, since the term "false sharing" implies unnecessary communication due to cache block granularity as originally coined by Goodman et al. [43], we use the term Update Silent Sharing throughout this thesis and encourage this usage throughout future literature. Communication eliminated by exploiting store value locality is due to properties of the data itself, not its spatial layout.

3.3 Temporal Silent Sharing

We introduced temporally silent stores in Section 1.4. To recap briefly, *temporal silence* describes a program-behavior phenomenon in which the net effect of two or more

writes changes a register or memory location to an intermediate value, but subsequently reverts that location to a previous value of interest. Since we focus on the value locality of store operations in this thesis, we are concerned with temporal silence of memory writes, or *temporally silent stores*.

Strictly speaking, we consider a store temporally silent if it writes a value to a memory location which has ever existed there previously. With this definition, consider a pathological case in which the same byte in memory is incremented repeatedly. After 256 writes of the location, all subsequent stores to the location become temporally silent, just with respect to a different previous value. Obviously such a definition does not have much practical value. Intuitively, we expect that recent previous values are more amenable to capture with fixed storage. Therefore, we consider a more restricted definition of temporal silence in this thesis, which we call *recent temporal silence*. In general, the *recent* modifier is meant to imply that only values which have existed in the near past are candidates for exhibiting temporal silence, with recent being defined by the desired application.

In a multiprocessor system, arguably the most interesting temporally silent stores are the ones that cause a memory location to revert to a value that has been previously observed by a remote processor. In this context, we consider a store *recently temporally silent* if it writes a value matching a stale value in a remote processor cache that is currently in invalid state or data previously written back to memory, implying that the candidate set of values exhibiting recent temporal silence is bounded by the number of processors in the system. This worst-case number of values occurs when each processor has previously observed different values prior to invalidation and memory is not up-to-date. These recent temporally silent stores change the location back to a value recently

observed by another processor or memory. We can imagine this value can be communicated at low cost. It will become clear shortly when rigorously redefining multiprocessor sharing why we choose such a definition. Note that there is nothing in our subsequent definitions which precludes multiple (distinct) intermediate values or temporally silent values for a given memory location; bounding the values is simply a matter of practicality. We use this definition of recent temporal silence throughout this thesis; we refer to it as simply *temporal silence* and stores writing such values as simply *temporally silent stores* for brevity.

As we have discussed, temporal silence has an important difference over update silence; namely an intermediate value which is non-update silent. Therefore, to succinctly describe specific dynamic stores of interest, we define a *temporally silent store pair* to consist of two parts: A non-update silent store to the memory location, called the *intermediate value* store, which visibly changes the system state from a previously observed value; and a non-update silent store, called the *temporally silent* store, which then reverts the state back to a previously observed value[1]. Note that there may be additional intervening stores to the same address; we do not consider these part of a temporally silent store pair. This definition is similar to the "silent store-pairs" exploited in Rajwar et al.'s work on Speculative Lock Elision [93, 90], except their definition stipulates that only atomic temporally silent pairs (Figure 3-1(c), to be discussed shortly) can be "silent store-pairs".

A common example of temporally silent store pairs in multiprocessors exists with lock variables, which assume an intermediate value when acquired and become tempo-

1. The term "pair" is a slight misnomer because the intermediate value store can consist of multiple dynamic store operations and multiple intermediate values, but this is consistent with Rajwar et al.'s terminology.

rally silent when released. Such an example is shown in Table 3-1. Note that the load at T5 is returning the value zero, as was previously seen by CPU 0 upon its last access to address A (at T1), thus the Read/Miss at T5 is not contributing any new information about the state of the system to CPU 0's execution. This is so because of the temporally silent store occurring at T4 on CPU 1. Note that no single dynamic store operation in the example is update silent, but the intermediate value store (T2) and the temporally silent store (T4) together yield a temporally silent store pair. Each part of the store pair is labeled in the example.

Table 3-1: Example of Temporal Silent Sharing In a Multiprocessor System. Assume sequential consistency and that Time indicates the order observed in the running system. The columns indicate instructions performed by each CPU as well as the transactions observed in a traditional multiprocessor system in response to these requests. The Intermediate Value Store and the Temporally Silent Store are indicated in the example with double outlines.

	CPU 0		CPU 1	
Time	Instruction	Cmd/Txn	Instruction	Cmd/Txn
T0	ST [A], 0	ReadX/Miss		
T1	LD [A]	Hit	LD [A]	Read/Miss
T2			ST [A], 1	Invalidate
T3			ST [A], 2	Hit
T4			ST [A], 0	Hit
T5	LD [A]	Read/Miss		

Having introduced the definition of temporally silent stores used throughout this thesis, we can return to a discussion of Figure 3-1. We can exploit temporal silence to further increase cache line lifetime. Two scenarios of interest are indicated in Figure 3-1(c) and Figure 3-1(d). In scenario (c), the cache line is written with an intermediate value store, but a temporally silent store follows and reverts the line to a previous value. Memory consistency rules allow the stores to be collapsed into a single atomic event that is now effectively update silent, since the memory location contains the same value before and after the store pair has executed. Therefore, we refer to this temporally silent store pair as an *atomic temporally silent store pair.* Atomic temporally silent store pairs can be created by exploiting freedoms available within weak memory models [3] or through employing

speculation and the coherence protocol to detect mis-speculation conditions [93]. We will discuss such approaches at length in Chapter 5. When such methods prove ineffective or undesirable, we must also consider the final scenario indicated in Figure 3-1(d).

In scenario (d), the cache line is written with an intermediate value and consistency rules force this store to be ordered; hence there is a time window when the previous copy of the cache line is invalid. Later, a temporally silent store occurs causing the current copy to match a previous copy, extending the cache line lifetime further.

We can rigorously define multiprocessor sharing with consideration of temporally silent stores, in a manner similar to Section 3.2. We accomplish this by further refining the definition of essential misess to capture the case where cache line values revert to the value previously observed by another processor in the system (the changes are in *italics*):

Essential Miss: A cold miss is an essential miss. Also, if during the lifetime of a block, the processor accesses (load or store) *a different value than the last value observed by that processor for that block* since the last essential miss to that block, it is an essential miss.

In our new definition of essential misses, we establish that the **net effect** of all writes to the location of interest since a processor's last observation of the location must constitute new system state for the miss to be essential. Note that this definition makes no statement about how many distinct processors have written to a specific word (with intermediate value or temporally silent stores) or other places within the cache line—it only requires that the value of interest be the same as the last value observed for a given processor. The distinction will become clear as we describe different methods of capturing temporal silence in Section 5. Also note that our new definition of temporal silent sharing

(TSS) is a superset of USS.

3.4 Understanding the Difference Between PTS, USS, and TSS

The definitions of essential miss presented in Section 3.1, Section 3.2, and Section 3.3, which are the cornerstone of redefining multiprocessor sharing to consider store value locality, are very similar. In order to make clear the difference between the classifications, we present Table 3-2, which contains similar transactions to those shown in Table 3-1, except we have added an additional case of USS.

Table 3-2: Understanding the Difference Between PTS, USS, and TSS.

	CPU 0		CPU 1	
Time	Instruction	Cmd/Txn	Instruction	Cmd/Txn
T0	ST [A], 0	ReadX/Miss		
T1	LD [A]	Hit	LD [A]	Read/Miss
T2	ST [A], 0	Invalidate		
T3			LD [A]	Read/Miss
T4			ST [A], 1	Invalidate
T5			ST [A], 2	Hit
T6			ST [A], 0	Hit
T7	LD [A]	Read/Miss		

New value defined implies True Sharing under USS

No new value observed implies miss removed under TSS

No new value defined implies miss removed under USS/TSS

Under Dubois' classification, the miss at T3 is considered essential, and thus is PTS. However, since the store at T2 is update silent, under the same classification this miss (if observed during execution) would be considered non-essential, and therefore false sharing under both USS and TSS. Similarly, since the store at T4 creates a new value (is not update silent), it implies an essential miss under USS occurs at T7. However, since the store at T4 is simply part of a temporally silent store pair, the miss at T7 is non-essential under TSS. Therefore, if it is observed during execution, it would be considered false sharing under TSS using the classification scheme of Dubois et al.

However, as stated previously, it is undesirable to call misses eliminated when

exploiting store value locality “false sharing” because of the spatial layout connotation associated with this term. How do we rectify this dilemma? We can resolve the issue simply if we realize an implicit assumption in the classification scheme, namely, that only misses which are observed during execution are in fact classified. Incidentally, this is one of the simplifications used in the basic classification presented by Dubois et al. [35] which allows neglecting capacity and conflict misses. They simply assume that no such misses can occur during execution (by using infinite caches).

Therefore, when considering sharing misses under USS and TSS, we avoid the need to call PTS misses eliminated under USS and TSS false sharing by simply saying that the misses are eliminated from the execution. This can be easily understood for the USS example (T3) by envisioning the store suppression procedure for the store at T2; if a load and compare is issued for the store, no invalidate will appear in the system, and the miss is eliminated. Although we have not described an exact mechanism for eliminating TSS misses (we will in Chapter 5), a similar argument can be made for the miss at T7.

Rigorously, we would say that the miss at T3 is an USS/TSS miss and that the miss at T7 is a TSS miss. This implies that such misses present in the execution could be eliminated by designing a system that eliminates update silent stores and temporally silent stores, respectively. With this understanding, any misses still present under USS or TSS cannot be eliminated by exploiting these types of store value locality, and can then be classified as either true sharing or false sharing as the classification scheme dictates.

Although this description might seem unduly complicated, the distinction, removal, and proper classification of misses is straight-forward in practice. We simply design systems which can detect update silent stores and temporally silent stores and elim-

inate the misses caused by these events. We can then directly compare the number of true/false sharing misses eliminated between the baseline case and the system which removes update silent stores/temporally silent stores to determine the fraction of baseline true/false sharing misses which are USS/TSS. The alternative is to significantly complicate Dubois' classification scheme to enumerate all sub-cases, in our opinion, sacrificing clarity. To make the overlap between classification schemes clear, we present a Venn Diagram in Figure 3-2.

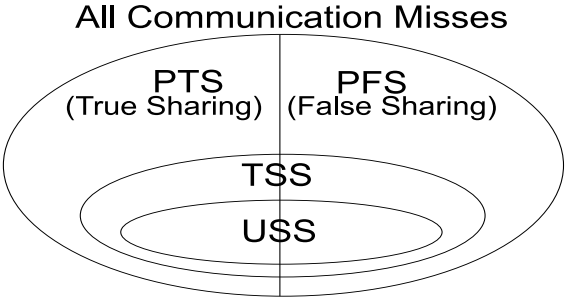

FIGURE 3-2. Venn Diagram of Communication Miss Classification. The USS and TSS ovals represent misses present in the baseline execution (without exploiting store value locality) which can be removed when exploiting update silent stores and temporally silent stores.

Given the previous discussion, we present data on communication misses removed with USS and TSS using the format shown in Figure 3-3. For now, we are only interested in how the data should be interpreted; the workloads this data was collected from are explained in detail in Section 3.6 and throughout subsequent chapters.

The left-most bars indicate reduction in communication misses for each scheme (for our four commercial workloads, presented in detail in subsequent chapters) with miss fractions normalized to the baseline case not exploiting any store value locality. The reduction in true sharing and false sharing from each simulation can be easily computed as the difference in height of each component between configurations. The right-most bars show that the reductions indicated would technically be classified using Dubois et al.'s

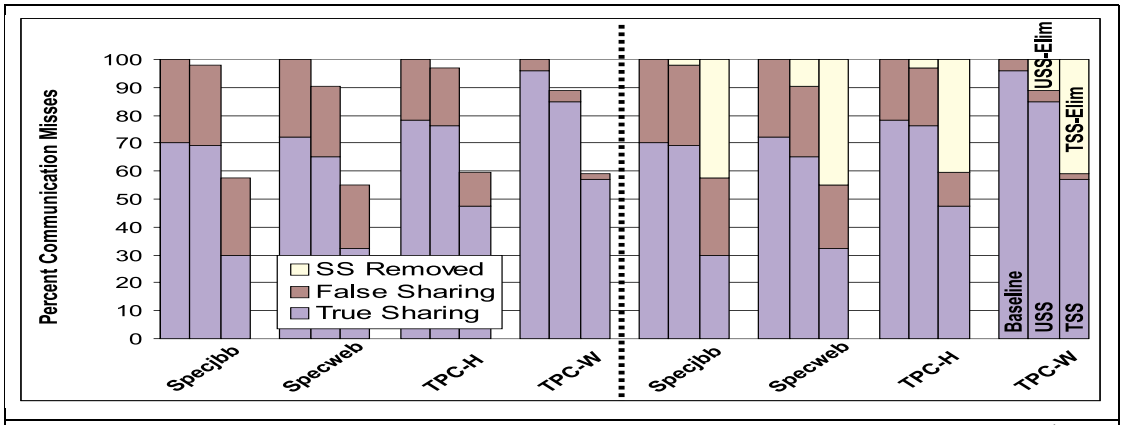

FIGURE 3-3. Example Presentation of Reduction in Communication with USS and TSS. The left-most bars indicate reduction in communication misses for each scheme with the communication miss fractions normalized to the Baseline case. The right-most bars indicate that the removed misses are technically considered USS and TSS misses, and therefore would be counted as "false sharing" misses using Dubois et al.'s classification scheme. For clarity, we present all results within this thesis in the manner of the left-most bars, with the right-most representation being equivalent and implicit.

scheme and would be considered non-essential misses. The two graphs are exactly equivalent—we choose the left-most presentation throughout this thesis, with the right-most representation implicit. Note that the right-most representation may split the USS-Elim and TSS-Elim components into true share and false sharing eliminated over the baseline—this fraction is easily computed as described previously.

3.5 Related Work

Dubois et al. [35] provided the most accurate definition of false sharing in multiprocessors based simply on address information. In their definition of false sharing, they also proposed hardware support to detect false sharing, allowing a remote processor to continue using a cache line that had been invalidated (without fetching the data from the previous owner) as long as the data it consumed had never been written in the aggregate by other processors. However, we assert that the key idea behind this scheme is trying to extend the usable lifetime of a cache line as long as possible (as illustrated in Figure 3-1). We would like to do this to avoid communication misses which are either due to artifacts of real systems, i.e. large coherence units, or intrinsic communication (true sharing).

Dubois's method focuses only on the former. Many other prior studies [36] [23] [34] also strive to extend cache line lifetimes by exploiting the freedoms granted by relaxed consistency models by delaying sending and processing of invalidation messages or similar techniques.

However, this work differs substantially from these because it examines the nature of the communicated values themselves. The exploitation of store value locality and usage of these other methods is largely orthogonal and we will treat it as such. However, particularly relevant interactions with store value locality will be detailed.

Speculative Lock Elision [93], proposed by Rajwar and Goodman, elides "silent store-pairs" which are very similar to our temporally silent store pairs. They use the elision and atomicity guaranteed through the coherence protocol as enablers to execute non-overlapping critical sections concurrently and also achieve the benefit of eliminating lock transfers and therefore cache misses. We have classified this method in Figure 3-1(c). SLE eliminates many TSS misses for the code idioms they target. Key differences between this thesis and their work are we propose methods for exploiting temporal silence without requiring recovery and machine roil-back support which can be implemented outside the processor core, provide a rigorous definition (in terms of sharing) of all misses which can be eliminated by exploiting store value locality, and also perform detailed evaluation with commercial workloads. We will return to a discussion of SLE and this thesis work in Chapter 5 and Chapter 6.[2]

Follow-on work related to exploiting common synchronization primitives co-pub-

2. The SLE work as well as this thesis research both took place while Ravi Rajwar and I were students at the University of Wisconsin. In my opinion, SLE and the foundation of this thesis were invented contemporaneously and independently.

lished with foundational work for this thesis includes Transactional Lock Removal [94] and Martinez et al. [77].

3.6 Simulation Environment for Exploration Studies

Throughout Chapter 4 and Chapter 5 we perform characterization studies and performance estimation from exploiting store value locality in multiprocessor systems. Given the complication in evaluating multithreaded workloads, particularly commercial workloads (described in detail by Alameldeen et al. [7] as well as our own work [67]), we perform detailed characterization studies using a full-system simulation environment with an in-order processor model including memory system timing based on SimOS-PPC [56], and later evaluate the most promising techniques from these initial studies in a full-system, execution driven, fully integrated timing simulation environment (Chapter 6). Starting with this simpler environment allows characterizing the intrinsic program behavior free of multithreaded workload non-determinism effects, as well as the ability to simulate longer intervals of execution because of the difference in execution throughput of the simulators. We choose metrics most closely related to the phenomenon of interest, e.g. miss rate, data traffic, address traffic, etc., to provide first-order indications of the possible benefits.

Unless stated otherwise, the basic machine configuration used throughout these characterization studies is shown in Table 3-3.

Note that the processor core used for these studies is similar to a single-threaded IBM RS64-III (Pulsar) [13] used in IBM S85 server systems. The perfect L1 cache CPI of 1.0 is very close to the measured CPI on real hardware for a similar system for these workloads [18]. We simulate four benchmarks from the SPLASH-2 [107] suite for entire runs

Table 3-3: Simulator Configuration for Characterization Studies. This configuration is used throughout Chapter 4 and Chapter 5 unless indicated otherwise.

Attribute	Value
Perfect Core IPC (L1 cache hits)	1.0
L1-I and L1-D Caches	128KB each, 2-way, 64B lines (1)
L2 Cache (unified)	16MB, 4-way, 64B lines (7)
Address Network	5 cycles occupancy, 45 cycles round-trip latency
Data Network (Cache to Cache Transfers)	70 cycles latency, 1 cycle occupancy (cross-bar)
Memory (DRAM)	140 cycles latency, 1 cycle occupancy
Operating System	AIX v4.3.1

Table 3-4: Benchmark Characteristics and Description. Instructions are measured excluding the operating system idle loop.

Program	Instr.	Loads	Stores	Description
barnes-4p	1.76B	410M	304M	SPLASH-2 N-body simulation (8K particles)
barnes-16p	1.89B	417M	304M	SPLASH-2 N-body simulation (8K particles)
ocean-4p	561M	177M	36M	SPLASH-2 Ocean simulation (258x258 ocean)
ocean-16p	1.41B	405M	81M	SPLASH-2 Ocean simulation (514x514 ocean)
radiosity-4p	2.43B	620M	326M	SPLASH-2 Light Interaction application (-room -ae 5000.0 -en 0.050 -bf 0.10)
radiosity-16p	2.53B	579M	307M	SPLASH-2 Light Interaction application (-room -ae 5000.0 -en 0.050 -bf 0.10)
raytrace-4p	414M	122M	45M	SPLASH-2 Raytracing application (teapot)
raytrace-16p	697M	228M	66M	SPLASH-2 Raytracing application (car)
specweb-4p	4.60B	1.13B	624M	Commercial Web-Serving application
specjbb-4p	3.58B	713M	431M	Commercial Server-Side Java application
specjbb-16p	723M	148M	88M	Commercial Server-Side Java application
tpc-h-4p	5.05B	454M	551M	Commercial decision support (TPC-H query 12, 2x)
tpc-h-16p	2.51B	533M	203M	Commercial decision support (TPC-H query 12, 1x)
tpc-w-4p	1.41B	336M	340M	Commercial Web-Based OLTP application (shopping)

of reduced input sets and three, three-second snapshots of commercial workloads described by Cain et al. [18] as well as an end-to-end run of TPC-H query 12 [106]. Table 1-1 provides a description of the benchmarks used. The SPLASH-2 applications were compiled with gcc version 2.95.2 for AIX at -O4 with alignment optimizations at 64 byte boundaries to mitigate false sharing. The commercial applications run various commercial databases, JVMs, etc. [18]. The input parameters for the SPLASH-2 programs as well as basic characteristics are shown in Table 3-4.

Note that benchmarks across different numbers of processors are not directly com-

parable. In most cases the input sets have been changed/scaled to execute sufficient instructions per processor to avoid cold-start effects in the case of the scientific workloads (which are end-to-end runs); The commercial workloads can be executing a completely different set and number of transactions both due to non-determinism effects [7] [67] and similar scaling issues. *Specjbb-16p* runs for a fixed transaction count, and both *tpc-h* runs are end-to-end; the other commercial workloads are snapshots of steady state execution. When measuring characterization data, i.e. cache miss rates, etc., snapshots of steady state execution are sufficient—we are basically performing a trace based analysis which is directly comparable between machine configurations. However, snapshots of steady-state execution complicate performance analysis of multi-threaded workloads (as detailed by Alameldeen et al. [7] and in our own work [67]). We introduce a method for performance analysis in this environment in Chapter 6, detailed in our own work [67].

Chapter 4

Update Silence in Multiprocessors

In our examination of store value locality in multiprocessors, we focus first on exploiting update silent stores. The problem of efficiently detecting and exploiting them in uniprocessors was covered in Chapter 2. Detection mechanisms discussed there are extensible in a relatively straight-forward way to multiprocessors. However, the data traffic performance potential as well as other multiprocessor implementation and system level issues will be discussed in this chapter. We use the multiprocessor benchmarks from Table 3-4 and the simulation environment described in Section 3.6 unless stated otherwise, for all characterization data.

4.1 Motivation and Background

We have shown in Table 1-1 that a significant fraction of dynamic store operations are update silent. Furthermore, throughout Chapter 2 we saw many scenarios under which we can exploit update silent stores for performance benefit in uniprocessor systems. However, additional benefit from update silent stores might be obtained in cache-coherent multiprocessor systems. In single-writer, invalidation-based cache coherence schemes, performing a dynamic store requires an exclusive copy of the cache line before the write can be performed. Similarly, stores are responsible for system-visible writes in update-based cache coherence protocols. Since many dynamic stores are update silent, we can eliminate many coherence events and data transfers related to stores in multiprocessor systems. Given the observation by industrial practitioners and academic researchers alike that communication-induced cache misses are a pressing performance limiter in modern systems, especially running commercial workloads, eliminating coherence and communica-

tion traffic will likely prove fruitful. We have discussed these issues at length in Section 3.1, as well as the basics of invalidation-based cache coherence in Section 1.1. Furthermore, previous to the work detailed in this thesis, all true-sharing communication misses were considered inherent and fundamental in multi-threaded program execution regardless of cache capacity. Data in this section will show that a non-trivial fraction of these true-sharing communication misses are easily eliminated through suppressing update silent stores.

4.2 Potential for Data Transfer Elimination

It should be obvious given the discussion in Section 3.1 that we can exploit update silent sharing (USS) to eliminate some fraction of communication misses. In order to explore this hypothesis, we implemented the measurement algorithm of Dubois et al. [35] with a MESI protocol [31], exercising it with a combination of commercial and scientific workloads for PowerPC under three different scenarios:

- The baseline scenario corresponds to Dubois' definition of sharing, classifying misses as either cold, false sharing, or true sharing throughout the benchmark's execution.
- The second scenario (USS-Hit) implements update silent store suppression, which effectively converts update silent stores into loads (Section 2.2). Note that from a multiprocessor cache perspective, a suppressed update silent store requires neither exclusive ownership of the cache line (as in an invalidate-based cache protocol) nor remote propagation of the updated store value (as in an update-based coherence protocol) because the value at the location of interest has not in fact changed. In this scenario, only cache hits are suppressed because of the potential effect sup-

pressing misses has on coherence transactions (discussed in detail in Section 4.3 and Section 4.4.) Therefore, we call this scenario USS-Hit.

- The third scenario (USS-P) measures the potential of USS with perfect knowledge of store update silence for both cache hits **and** misses. Cache hits are handled as described in the second scenario (USS-Hit). For cache-missing stores, if the store is known to be update silent, a Read (sometimes called a Get [31]) transaction is performed to obtain the data with sufficient permission to perform only the load portion of a store-verify, knowing the store will be suppressed. If the store is known to be non-update silent, a Read-Exclusive (ReadX, sometimes called a read-with-intent-to-modify or GetX [31]) is performed to obtain the data with write permission immediately, avoiding an additional coherence transaction to upgrade the line. Note that this scenario indicates the maximal data-traffic benefit of eliminating update silent stores from the observed execution. In subsequent sections we normally refer to this scenario simply as USS (instead of USS-P) for brevity as it indicates the best possible data traffic performance of exploiting update silent stores.

Results of these simulations are illustrated in Figure 4-1, with the three scenarios described previously for infinite caches and a 4MB 8-way associative cache. Focusing first on the communication miss components, we see a measurable reduction in both true and false sharing in *ocean*, *radiosity*, *specweb*, and *tpc-w*. In *barnes, raytrace, specjbb,* and *tpc-h* there are still many dynamic update silent stores (40%, 38%, 35%, and 72%, respectively, shown in Table 1-1), but suppressing them does not lead to a tangible communication miss reduction. This can be due to non-update silent stores to other locations

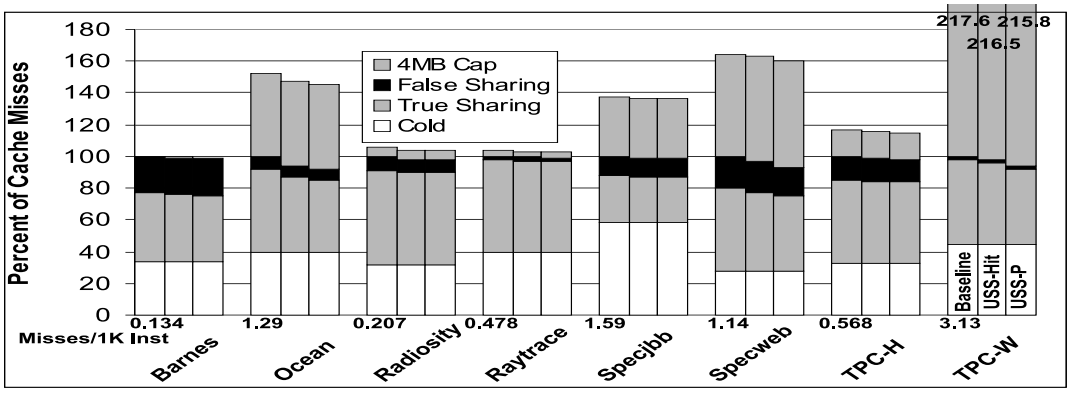

FIGURE 4-1. Percentage of Cache Misses for Different Definitions of Sharing. The data is normalized to the Baseline case for 64B cache lines and infinite caches (unified instructions/data). The contribution of additional capacity/conflict misses for an 8-way associative 4MB cache is also shown.

either truly or falsely shared within a cache line or because the update silent stores are occurring to unshared locations. For these benchmarks, we also examined the effect of reducing cache line size and the relative fraction of avoidable communication misses stays constant—indicating that for these workloads the phenomenon is likely the latter.

It is also interesting to examine the effectiveness of update silent store removal for finite caches, with data for a 4MB, 8-way set associative cache also in Figure 4-1. As expected, the overall miss rate reductions are not as large as the reductions in sharing misses. This is due to two effects: 1) Finite cache capacity necessarily increases the number of misses, some of which are also true or false sharing communication through memory; 2) The increased working set brought about by reduced invalidations in the system (affecting replacement of lines that would've previously been marked invalid in remote caches but are no longer because of update silent store suppression). This second effect is most readily observed in *tpc-w*, where we see a measurable reduction in communication misses, but we see no appreciable reduction in overall miss rate with a finite cache. The only explanation is that communication misses have been replaced by capacity/conflict misses. However, the infinite cache results indicate that the amount of communication can

be reduced by exploiting update silent stores. Therefore, the working set has either been increased by update silent store suppression or the replacement policy is non-optimal when update silent stores are suppressed.

In *ocean* and *specweb* we see measurable reductions in overall miss rate for finite caches, indicating that exploiting update silent stores for data traffic reduction is worthwhile. Furthermore, overall data traffic is additionally reduced in all benchmarks because suppressing update silent stores also eliminates writebacks. The writeback reduction rate for each configuration is shown in Figure 4-2. The writeback rate per 1K instructions is also shown to indicate the writeback behavior intrinsic within the benchmark for a 4MB cache.

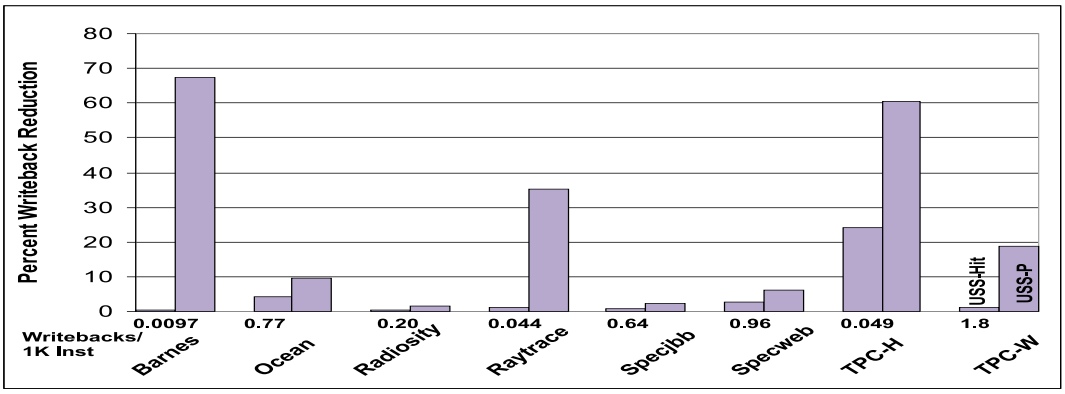

FIGURE 4-2. Percentage of Writebacks Removed By Suppressing Update Silent Stores. The data is normalized to the Baseline case for 64B cache lines with an 8-way associative, 4MB cache. Writeback reductions for both USS-Hit and USS-P silent store squashing are indicated.

We can see that substantial writeback reduction is only obtained in *tpc-h* with USS-Hit update silent store suppression. However, when applying USS-P update silent store suppression, we see substantial writeback reduction in *barnes*, *raytrace*, *tpc-h* and *tpc-w*. This indicates that a substantial fraction of store misses are update silent, and therefore, suppressing update silent store misses is important for eliminating writebacks. This can be easily accomplished using either of two methods: 1) Issuing a ReadX transaction

on the miss, comparing the written value against the value returned by the memory system and conditionally installing the line in either exclusive (E) or modified (M) state, or 2) Issuing a Read transaction on the miss, comparing the written value against the value returned by the memory system, and conditionally sending a ReadX to upgrade to modified (M) state when a store miss is non-update silent.[1]

The writeback rate in *barnes*, *raytrace*, and *tpc-h* is low—less than 1 writeback every 20K instructions—implying that relatively little data traffic might be saved by eliminating writebacks. However, when comparing with Figure 4-1, we see that although the writeback rate is low, in *tpc-h* and *tpc-w* we actually save approximately two times as much data traffic from writeback removal with USS-P compared to sharing miss reduction. This occurs because the fraction of writebacks removed is much larger as compared to the fraction of sharing misses removed for USS-P.[2] Therefore, non-trivial data traffic reduction can be achieved exploiting update silent stores for both communication and writeback reduction.

4.3 Address Traffic Considerations

We shift our focus to the effect update silent store exploitation has on address traffic in a MESI-based multiprocessor system. In Figure 4-3 we indicate the incoming invalidate rate (both hit and miss) for the three cases outlined in Section 4.2 for a 4MB 8-way associative cache. The stacked bars show the rate at which invalidates (including those triggered by both store clean hits and store misses) hit in a remote cache, miss in a remote

1. Each of these approaches has a different cost/benefit in terms of sharing miss reduction, as well as store commit latency. We defer discussion of handling store misses in the context of update silent stores until Section 4.3 and Section 4.4.
2. The fraction of writebacks removed does not include “implicit writebacks” required in MESI on dirty misses for Read transactions. We only calculate this reduction as explicit modified (M) lines replaced from the cache.

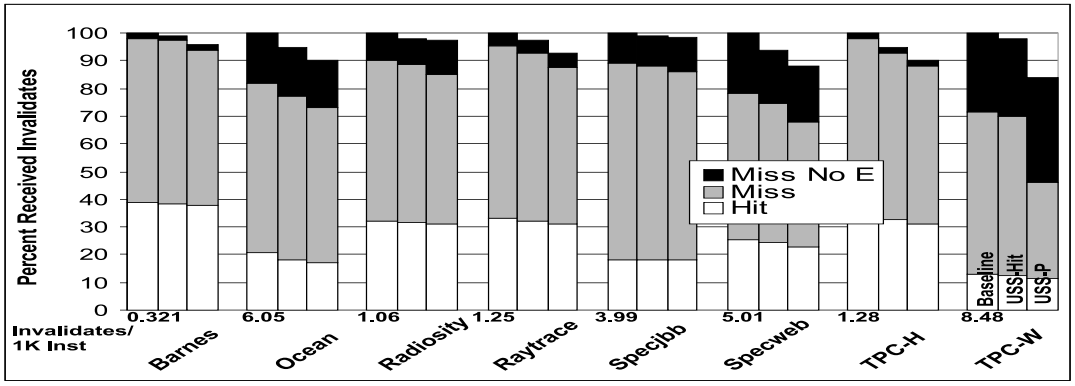

FIGURE 4-3. Multiprocessor Invalidation Reduction by Exploiting Update Silent Stores. Received invalidates which hit a remote cache or miss a remote cache for Baseline, USS-Hit, and USS-P are shown. Results for a MESI protocol and additional invalidates which would occur in an MSI protocol are also indicated for a 4MB, 8-way associative cache.

cache, and miss in a remote cache if E state is not implemented (hence invisible E->M upgrades are not possible, and an explicit upgrade request from S->M is required.)

For all benchmarks, we see a measurable reduction in invalidate traffic even for simple USS-Hit store suppression. Note that in general, we observe greater reductions in invalidates which miss remote caches, indicating that silent store suppression is most effective at eliminating so-called “useless” invalidates. Since address bus bandwidth is a precious commodity in snoop-based SMPs, this is a desirable property.

Note that reduction in required address bandwidth is greatest in USS-P. However, as mentioned in Section 4.2, USS-P requires oracle knowledge of store update silence at the time of a store miss so either a Read or ReadX can be issued to fetch the cache line of interest. If Reads are issued for all store misses, all non-update silent store misses will necessitate two address transactions: 1) the initial Read; 2) an Upgrade (because the store is not update silent). This is obviously undesirable. If ReadXs are isued for all store misses, all update silent store misses will unnecessarily invalidate remote copies of the data, reducing the data sharing benefits of update silent store suppression (illustrated in Section 4.2.) Obviously, there is a trade-off between eliminating sharing misses (issuing

Reads followed by Upgrades for non-update silent store misses) versus reducing address traffic and store commit latency (always issuing ReadXs for store misses).

To examine the trade-off between these two differing approaches, we explore store update silence for misses where the data is not already present in the cache, i.e. true store misses, not store-clean misses which only lack exclusive permission to write the line but the data is already cache-resident. The percentage of such update silent store misses is indicated in Figure 4-4 for both a finite (4MB, 8-way associative) and infinite cache.

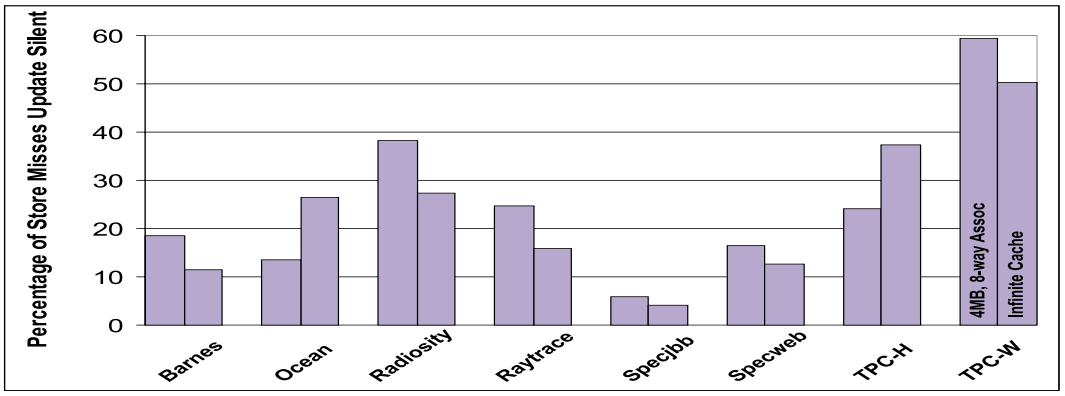

FIGURE 4-4. Percentage of Update Silent Store Misses. The percentage of store misses without data (true store misses, not upgrades for solely exclusive ownership) which are contributed by update silent stores for both a 4MB, 8-way associative cache and infinite cache are shown.

We can see from the figure that the fraction of update silent store misses is largely dependent on benchmark (ranging from 5% in *specjbb* to almost 60% in *tpc-w*) and no definite trend emerges based on different cache sizes across benchmarks[3]. However, in all cases (except *tpc-w*) less than 40% of store misses are update silent, indicating that handling store misses by a Read followed by an Upgrade for non-update silent store misses is most likely not the best choice; this policy will substantially increase address transactions in the system as the majority of store misses will now require two transactions instead of

3. This can be due to various factors, including truly shared data communicated through memory due to finite cache capacity.

one.[4] Furthermore, handling store misses in this way will likely slow the processor core's ability to commit store operations because of the additional latency for the Read/Upgrade as compared to the ReadX in the base case. We will revisit the impact of store commit throughput in Chapter 5 when discussing temporal silence and also in our performance evaluation in Chapter 6.

As described, handling store misses with a Read/Upgrade pair seems undesirable. However, as shown in Figure 4-1, simply issuing a ReadX for store misses foregoes opportunity to eliminate USS misses. Therefore, we must work a little harder to capture the available opportunity. Options at our disposal include:

- Predict which stores are likely to be update silent. To provide a more targeted prediction, we may focus only on those stores which normally contribute to communication misses instead of all stores. Program structure-based predictors for update silence may be useful, as well as cache information for tag-match but non-writable status for the cache lines of interest may aid in training.
- Provide a distributed mechanism for deciding whether a store miss is likely to be update silent. We can imagine sending the store miss data (potentially in a compressed form) along with a special *Conditional ReadX* transaction. When the request arrives at the current owner of the cache line, the owner can compare the store data, or some other attribute, with the current data value to determine the likelihood of store update silence. Based on this outcome, the current owner provides (through an appropriate snoop response) a shared copy of the cache line to

4. Note, however, that the actual address transaction increase will be much smaller than the fraction of non-update silent store misses due to other transactions observed in the system such as load misses, writebacks, cache line operations like the PowerPC data-cache-block-zero which occurs frequently in our workloads, etc.

the requestor if the store is likely update silent or an exclusive copy otherwise.[5]

- A final option is to provide a facility for issuing the ReadX for the store miss, but allow the requestor to re-install the cache line into remote processors if update silence is determined and exploiting update silence will likely lead to a remote miss being prevented. We call such silence *useful silence*; it is useful in the sense that exploiting it eliminates a dynamic instance of sharing. We describe a protocol enhancement which enables this mechanism of exploiting useful update silence when discussing protocol support for temporal silence (Chapter 5). Therefore, we will return to a discussion of handling update silent store misses using this mechanism after describing the protocol enhancement and related material, i.e. Section 5.7.4.

Each solution may be applicable, depending on system-level constraints. We detail the method used in our performance studies, and justification for it, in Section 5.7.4. We present execution-time improvement through exploiting update silent stores in multiprocessors in Section 6.4.

4.4 Critical Update Silent Stores

We have shown that update silent stores can be exploited to eliminate a substantial fraction of both data and address traffic. However, we also explained in Section 4.2 and Section 4.3 potential complications in handling both store hits and store misses to minimize system-address traffic, leading to two different policy studies (USS-Hit and USS-P).

We introduced the concept of *critical update silent stores* in uniprocessors in Sec-

5. The utility of this technique is dependent on the interconnect model assumed for the system. In bus-based, snooping, multiprocessors such a technique may be envisioned; in directory systems or systems without appropriate snoop-response support, this technique might not be useful or might require special modification.

tion 2.4.1.2, indicating that it is sufficient to suppress only the critical update silent stores to achieve maximal writeback reduction. As a brief reminder, we call an update silent store critical if suppressing it will lead to a reduction in writebacks; not all update silent stores are necessarily critical due to possible interactions with non-update silent spatial and temporally local stores during a cache line lifetime. As indicated in that discussion, the notion of criticality has expanded meaning, as well as additional relevance, in multi-processor systems, beyond the simple difference described for USS-Hit and USS-P in Section 4.3.

Assume for the moment that we are only interested in suppressing update silent stores to reduce data traffic and address traffic between processors, as discussed in this chapter, and not the processor core benefits described in Chapter 2. In this coherence-only context, the natural store atom to consider is no longer a single dynamic store, but rather an entire cache line. Considering this, naively suppressing all update silent stores may be wasteful due to many factors. For example, suppressing update silent stores to a cache line which has already entered modified state is wasteful because the line is already exclusive in the current cache and is also dirty and will need to be written back; no data or address traffic reduction will result. Therefore, in the context of coherence transactions, this is a spatial locality aspect of efficient update silent store suppression. Furthermore, a temporal aspect should be considered as well. To illustrate this temporal aspect, let's explore address transactions in the MESI protocol when suppressing update silent stores with the oracle USS-P predictor for store misses. Practically, this means we issue either a Read or ReadX request on each store miss with perfect knowledge of whether the store causing the miss is update silent.

With this understanding, consider the following scenario: a processor encounters an update silent store miss, consults the oracle predictor, and issues a Read since the oracle informed it of store update silence a priori. Let's assume remote snoop responses indicate the line should be installed in shared state. If a non-update silent store is later performed to the cache line, it will require an upgrade to obtain write permission. Notice that two coherence transactions were required (one Read for the miss and an additional upgrade), in contrast to basic handling where the update silent store miss would have lead to a ReadX eliminating the additional upgrade. Note that the Read/Upgrade pair is not always detrimental to system performance; if suppressing the update silent store miss extended the cache line lifetime in a single remote cache sufficiently to eliminate a remote miss, an improvement in address and data traffic is achieved. In fact, this is the motivation for suppressing update silent stores in multiprocessors.

However, if a remote miss was not eliminated by suppressing the update silent store miss, less overall traffic would have resulted if the cache line was requested with a ReadX **even though** the initial store miss was update silent, avoiding the Read/Upgrade pair. The detrimental effect of the Read/Upgrade pair on system performance can be twofold: First, additional address traffic is created increasing queuing delays for other coherence transactions for all processors. Second, this additional upgrade causes commit of the non-update silent store to be delayed because it waits for write permission. This delay may be partially hidden through write buffering or exclusive prefetching but nevertheless, some of the upgrade latency may be exposed to program execution. Therefore, optimally exploiting update silent stores is not only a question of update silent store miss prediction, but critical update silent store miss prediction.

.

Table 4-1: Illustration of Critical Update Silence in Multiprocessors. An oracle update silent store miss predictor will cause a Read (instead of a ReadX) for the store at T1. However, this line is later upgraded for a non-update silent store. Since the suppressed update silent store does not eliminate a remote miss, suppressing it is anti-critical. The baseline execution is shown in the bottom half of the example.

	CPU 0		CPU 1	
Time	Instruction	Cmd/Txn	Instruction	Cmd/Txn
T0			LD [A] (0)	Read
T1	ST [A], 0	Read		
T2	ST [A], 1	Upgrade		
T3			LD [A] (1)	Read/Miss
Baseline Execution Without Update Silent Store Suppression				
T0			LD [A] (0)	Read
T1	ST [A], 0	ReadX		
T2	ST [A], 1	<None>		
T3			LD [A] (1)	Read/Miss

To illustrate the point more directly, we present Table 4-1. The execution with USS-P update silent store suppression is shown in the top part of the table, while the baseline execution without update silent store suppression is shown in the bottom. Notice the additional upgrade transaction at T2 in the USS-P scenario, indicated with the double outline. Since suppressing the update silent store miss at T1 (issuing the Read) did not lead to an eliminated sharing miss on another processor, issuing a single ReadX at T1 would've both reduced system address traffic and also reduced the store commit latency at T2.

With this understanding, we can define terms to describe the different scenarios of interest. In general, we use the term *critical* to indicate that suppressing this update silent store will prevent a communication miss, *anti-critical* to indicate that suppressing an update silent store in this scenario will cause an additional upgrade transaction not present in the baseline, and *non-critical* to indicate that suppressing an update silent store will have no effect because no additional upgrades over the baseline case are required. More rigorously, we define the following types of store misses considering update silent stores (we assume infinite caches for ease of discussion):

Critical Update Silent Store Miss: *An update silent store miss, which if sup-*

pressed, will prevent at least one dynamic instance of USS, i.e. will prevent a sharing miss, is a critical update silent store miss.

Such misses are critical because suppressing them leads to sharing miss reduction.

Anti-Critical Store Miss: *A non-update silent store miss which is not system-cold, i.e. the first reference to a cache line by any processor in the system, is anti-critical. Furthermore, an update silent store miss to data which is not accessed by at least one other processor before being written non-update silently is also anti-critical.*

Such misses are anti-critical because they are not update silent and at least one other processor has a copy of the data. If no other processor has a copy of the data (system-cold), the cache line would be installed in exclusive state and is therefore not anti-critical.

Non-Critical Store Miss: *An update silent store miss which is not accessed by another processor and also never written non-update silently is non-critical. Furthermore, an update silent store miss which is later accessed with only cold misses from other processors in the system before being written non-update silently is non-critical. Finally, any system-cold store miss, i.e. the first reference to a cache line by any processor in the system, is non-critical.*

These update silent store misses are non-critical because suppressing them neither leads to reduced communication nor a subsequent upgrade. System-cold misses are non-critical because they will be installed in exclusive state since they are, by definition of system-cold, not cached by any other processor.

We have previously discussed only store misses, i.e. no copy of the data is present before the store is being performed. In the case of store clean hits, i.e. a copy of the data is

present but in a non-exclusive state, simplifications to the above definitions can be made.
Anti-critical and non-critical have no distinction because store update silence can be verified using the local, non-exclusive, copy; if the store is update silent, our non-exclusive copy is maintained, if it is non-update silent an upgrade is performed. In the case where a subsequent non-update silent store writes the line without preventing a sharing miss, the upgrade has simply been deferred, but an extra address transaction is never created. Therefore, we refer to such update silent stores as non-critical update silent stores. However, critical update silent stores still exist for store clean hits, which we formally define as:

Critical Update Silent Store: *An update silent store, which if suppressed, will prevent at least one dynamic instance of USS, i.e. will prevent a sharing miss, is a critical update silent store.*

It is critical because suppressing it leads to sharing miss reduction. Note that stores which create critical update silent store misses are also critical update silent stores.

Table 4-2: Illustrating the Classification of Store Misses. All true store misses are indicated with double outlines. Update silent stores are shown with (*). Address transactions observed in a machine performing oracle USS-P update silent store suppression are shown. Assume [A] contains the value zero and misses all caches to begin. The store misses with double outlines are non-critical, critical, and anti-critical from top to bottom.

	CPU 0		CPU 1		CPU 2	
Time	Instruction	Cmd/Txn	Instruction	Cmd/Txn	Instruction	Cmd/Txn
T0					LD [A] (0)	Read
T1	ST [A], 0*	Read				
T2			LD [A] (0)	Read		
T3			ST [A], 1	Upgrade		
T4					ST [A], 1*	Read
T5			LD [A] (1)	<None>		
T6					ST [A], 0	Upgrade
T7	ST [A], 0*	Read				
T8	ST [A], 1	Upgrade				
T9			LD [A] (1)	Read		

We illustrate each type of miss in Table 4-2. The table shows address transactions observed in a machine performing USS-P update silent store suppression which issues

Read transactions for update silent store misses. Assume [A] misses all caches and contains the value zero at time T0. Store misses are shown with double-outlines in the table. In this example, all store misses happen to be update silent store misses. The store miss at T1 is not critical; although it is accessed by another processor at T2, since the access at T2 is a cold miss, suppressing the update silent store at T1 achieves no sharing miss reduction. Since a non-update silent store has not occurred between T1 and T2, the store miss at T1 is also not anti-critical. Therefore, it is non-critical, i.e. issuing either a Read or ReadX at T1 would have been inconsequential[6]. The store miss at T4 is critical; suppressing this update silent store directly avoids a sharing miss at T5. The store miss at T7 is anti-critical; we observe the Read/Upgrade pair at T7 and T8 which is characteristic of anti-critical.

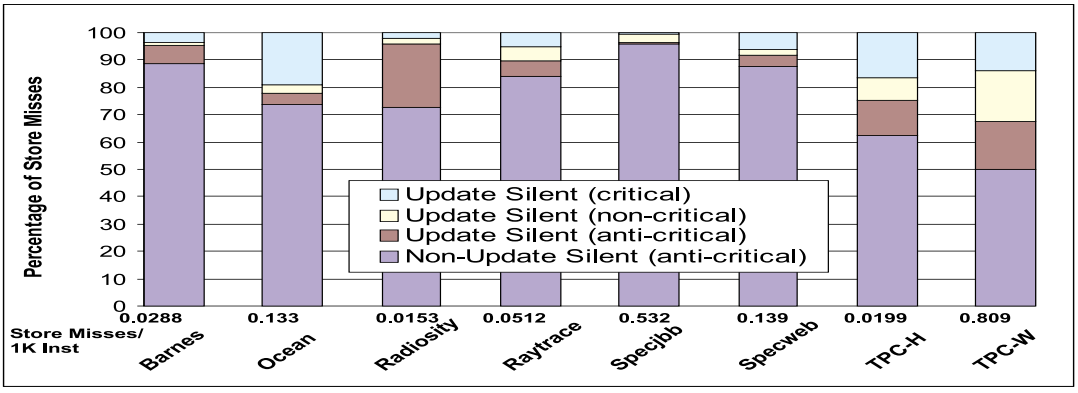

FIGURE 4-5. Criticality of Store Misses. The percentage of store misses without data (true store misses, not upgrades for solely exclusive ownership) which are anti-critical, non-critical, and critical are indicated. The large sub-fraction of anti-critical due to non-update silent store misses is indicated separately.

We show the fraction of critical, anti-critical, and non-critical store misses using the classification scheme described for infinite caches in Figure 4-5. We see that the majority of anti-critical store misses are non-update silent store misses. However, these

6. We are assuming a broadcast-based, snooping, protocol. In a directory protocol, miss latency may be reduced by suppressing this update silent store to convert the cold miss at T2 into a 2-hop instead of 3-hop miss [31]. Therefore, this miss is still critical in some sense in a directory-based system. However, cold misses are the only case where this scenario arises, so we neglect it for the sake of brevity.

are trivially anti-critical as they are non-update silent. We are principally interested in examining the critical, non-critical, and anti-critical behavior of update silent store misses (top three sections of the stacked bars) as this illuminates whether considering memory behavior beyond the update silent store miss event itself is worthwhile. We focus on the critical and anti-critical categories, because by definition, the non-critical category can be serviced with either a Read or a ReadX without negative impact and since the non-critical component is the dominant component of update silent store misses for only a single benchmark (*tpc-w*).

No definite trend exists across benchmarks in the fraction of update silent store misses which are critical versus anti-critical. In *ocean* and *tpc-h*, there are significantly more critical store misses, in *radiosity* and *tpc-w* the converse is true. The remaining benchmarks show a nearly equal distribution between the two types. Due to the substantial population of anti-critical update silent store misses in some cases, criticality should be considered, beyond simple update silence prediction, to minimize address transactions in the system. Also, due to the substantial population of critical update silent store misses, policies which do not suppress update silent store misses will likely sacrifice significant potential for communication reduction. This is evidenced in previous results comparing the simple USS-Hit policy against USS-P (Figure 4-1).

In order to gain further insight into the program behavior and also the predictability of criticality, we can explore the memory footprint of stores exhibiting critical behavior. Figure 4-6 shows the cumulative memory footprint of locations exhibiting USS avoidable misses and therefore, critical update silence. Scientific workloads are shown in the top part of the graph, commercial workloads on the bottom. In the scientific work-

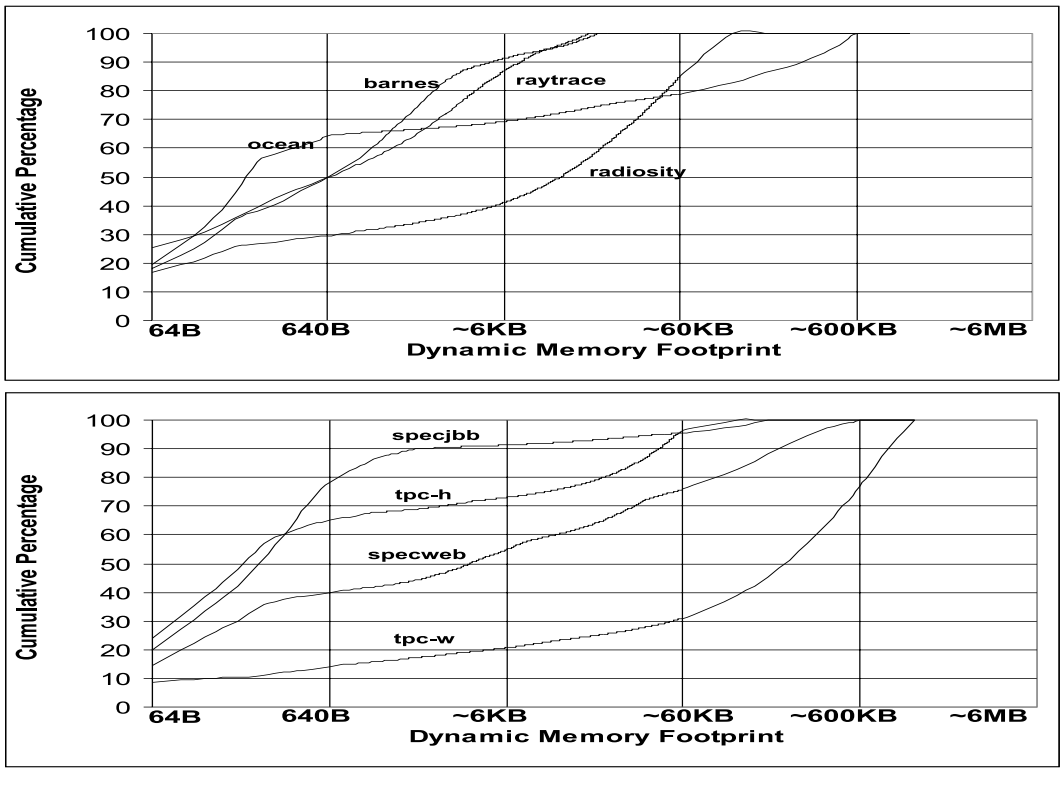

FIGURE 4-6. Dynamic Memory Footprint Contributing to Critical Update Silence. The cumulative distribution of memory addresses (measured at cache line granularity) contributing USS-avoidable misses is indicated. The scientific workloads are shown in the top graph, commercial workloads in the bottom graph. Notice the log scale on the x-axis for both graphs.

loads, we observe that approximately a 60KB working set of memory locations contributes over 80% of all USS avoidable misses, with *barnes* and *raytrace* exhibiting the smallest working sets; less than 6KB is needed to capture 80% of USS avoidable misses in these workloads. The commercial workloads exhibit a larger working set in general (nearly 1MB is required in *tpc-w* to reach 80% of opportunity captured, for example) and also more variation between workloads. This is reasonable given the observation by many of increased memory footprints of commercial workloads [96, 75, 9].

We also observe that the dynamic footprint of memory locations exhibiting critical update silence is proportional to the fraction of sharing misses which are eliminated under USS; in comparing Figure 4-1 with Figure 4-6, we observe that the three benchmarks ben-

efiting the most from USS-P update silent store suppression (*ocean*, *specweb*, *tpc-w*) also have the largest footprint of critically update silent memory locations. This indicates that most benchmarks reaping greater benefit under USS do so through sharing a greater number of memory locations update silently, as opposed to more frequently sharing the same locations update silently. Therefore, we must consider this large working set requirement when designing predictors to enable efficient exploitation of USS.

We describe a critical update silence prediction mechanism and handling of update silent store misses in Section 5.7.4 after detailing a protocol enhancement to exploit temporal silence. Update silent store misses can be handled seamlessly in our new coherence protocol, allowing both criticality prediction (as discussed in this section) as well as achieving baseline store commit latency for non-update silent store misses (as discussed in Section 4.3).

4.5 Memory Consistency and Correctness Implications

When discussing any memory optimization concerning visibility of memory values, it is important to consider the memory consistency model. In addition, memory operations may contribute changes to system state beyond the values they explicitly deliver into the memory hierarchy through implicit page table updates or other side-effects. In this section, we explore both the consistency and correctness issues relating to update silent store suppression.

4.5.1 Memory Consistency Considerations

We argue that exploiting update silent stores does not materially complicate multiprocessor system design because, in common consistency models [3], the store verify can be performed at an equivalent point with respect to consistency constraints as the store.

When the store verify is performed under these constraints, the store can safely be suppressed because it communicates no change in system state. However, in memory models which do not treat loads and stores equivalently, e.g. processor consistency, total store order [3], the hardware must be sure that the store verify (load) is constrained to store ordering rules when performing the store verify.

We reason about the correctness of update silent store suppression for sequential consistency [61] using the constraint graph approach which is discussed by Landin et al. [62]. At a high level, the constraint graph labels memory operations with directed edges indicating the observed order between memory references to the same address and also indicates other memory model constraints, such as program order, through additional directed edges. An execution can be shown to be correct if the constraint graph is acyclic. Proofs for weaker models follow similarly, and we will discuss them briefly after describing the solution for sequential consistency.

Sequential consistency (SC) states that all memory operations performed by all processors must appear to be interleaved into a single total order. If a total order of all memory operations cannot be constructed, the memory system does not provide SC. If we assume a memory system which guarantees SC as the baseline, the following constraint edges exist concerning stores in the memory constraint graph:

WAW: Write-After-Write

WAR: Write-After-Read

RAW: Read-After-Write

To prove suppressing update silent stores by the process of store verification is correct under SC, we must prove that the execution provides the appearance of a total order of

all memory operations. We start by taking an existing constraint graph, assumed to be correct for SC without considering update silent stores, with all constraint edges described as above (WAW, WAR, RAW). Then, we describe how existing constraint graph edges are converted in the presence of update silent store suppression. Finally, we argue correctness using two approaches: showing how constraint edges changed with update silent store suppression still impose the same execution or how changed edges which no longer impose the same execution are not required since they contribute no change to system state.

To perform the edge conversions for update silent store suppression, we essentially convert a store into a load; this is the process of store verification, as described in Chapter 2. To make the discussion brief, we assume the reader is familiar with the constraint graph representation [62]. An example constraint graph is shown in Figure 4-7. We describe the scenario of interest in the figure in detail later, however, examining the figure to make the constraint graph and edge conversions more concrete may be helpful.

Constraint graph edge conversions for each edge type are:

WAW: This edge may be converted into a RAW with update silent store suppression in the case where the source store contributing to the edge is non-update silent but the target is update silent. Since RAW and WAW impose the same constraints on execution under SC, the execution is unchanged. Similarly, if the source store is update silent, but the target is non-update silent, this edge may be converted into a WAR edge, which again imposes the same constraints on execution under SC, and the execution is unchanged. In the case where both contributing stores are update silent, this edge may be converted into a RAR, which we discuss separately.

WAR, RAW: These edges may be converted into RAR edges with update silent store suppression. Since RAR edges do not impose execution constraints under SC, this implies a different execution is possible. However, to prove suppressing update silent stores is still correct, we need only prove that the same execution will still be observed. We discuss such RAR edges next.

WAW, WAR, RAW Conversion to RAR: Guaranteeing these constraint graph edge conversions will create the same execution, and hence will be SC, is a bit trickier. However, once we realize that suppressed update silent stores inherit a RAW edge from the previous non-update silent store to the memory location of interest the conversion becomes simple. This edge is not shown in the baseline constraint graph because it was transitively reduced. This means it was present in the baseline execution but it was not required in the constraint graph to describe the execution; this *transitive RAW edge* was essentially hidden by a subsequent WAW, WAR, or RAW edge. This transitive edge dictates that the update silent store must be suppressed with the previous non-update silent value. An example of RAR edge conversion is shown in Figure 4-7, with original constraint graph edges for the observed execution shown in solid lines, and the transitive RAW edge introduced by update silent store suppression indicated with the dashed line. Edges converted from WAW into RAW and RAR by update silent store suppression, as well as the two update silent stores, are also indicated.

As stated, RAR edge conversion may remove previous edges constraining execution from the constraint graph. However, in previously published work on validating and verifying SC, it was shown that RAR edges can be safely ignored [27, 42]. This is intuitive—only instructions which create values observable through memory, i.e. non-update

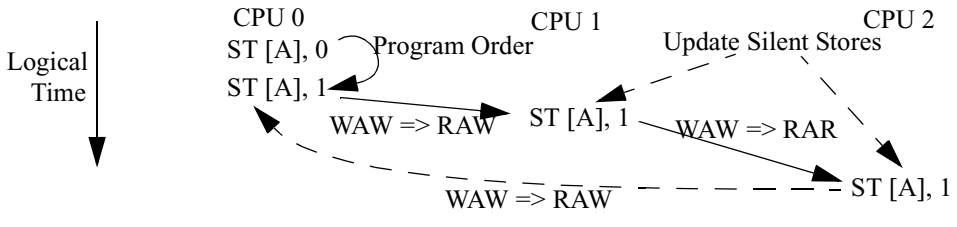

FIGURE 4-7. Illustration of Transitive RAW Edge in the Constraint Graph. The original constraint graph is shown in solid lines, along with the original constraint graph edge types. Update silent store suppression converts the two WAW edges into RAW and RAR edges. The exposed transitive edge revealed due to update silent store suppression is the dashed line.

silent stores, need strict ordering requirements with respect to other memory operations. A subtlety arises with update silent stores of course, since update silence can only be properly determined with respect to the previous non-update silent store in the global order. This is precisely the WAW edge converted into a RAW edge, as described above. We have already described the correctness constraint in this case.

Guaranteeing that all store verifies inherit this transitive RAW edge can be accomplished through various means. Fortunately, this problem has already been extensively studied in the context of high-performance out-of-order processors which implement SC. As a simple example, consider the speculative load hoisting approach used in the MIPS R10000 [110]. The same mechanism can be used for speculative store-verifies to ensure verification with the previous non-update silent value. Additional methods, as well as a detailed discussion is available in Gharachorloo's thesis [40].

The conditions for other common memory models (processor consistency, total store order, and weak order [3]) are similar. Constructing the constraint graph for these models is discussed by Cain et al. [16] [17]. The key requirement for correctly suppressing update silent stores is to ensure that the store verify (load) must adhere to store ordering constraints with respect to the consistency model. Under memory models which handle loads and stores with similar constraints (sequential consistency, weak order), using exist-

ing unmodified hardware and simply issuing a load in place of the update silent store may be viable as is done by Kim and Lipasti [57]. However, in processor consistency and total store order, loads and stores exhibit different ordering semantics, implying that a special type of load operation (with store ordering requirements but which does not obtain an exclusive copy of a cache line) may be required.

4.5.2 Existing ISA Correctness Considerations

We have argued in the previous section that exploiting update silent stores does not materially complicate multiprocessor system design because, in common consistency models [3], the store verify can be performed at an equivalent point with respect to consistency constraints as the store.

However, in practice, the previous statement only holds for stores which have no side-effects other than communicating new values. However, stores ***can*** have other side-effects beside delivering a new value into the memory location being written. For example, most architectures define certain memory references to be either cacheable or non-cacheable for various reasons, e.g. I/O space memory ranges, explicit I/O writes. Any non-cacheable memory location should avoid update silent store suppression to maintain correct operation. Such non-cachable references are typically indicated either through page permissions, range checking, or explicit instruction encoding, thus avoiding store verification or suppression for such writes is straight-forward.

Additional complications arise for cacheable memory locations, and we strive to illuminate such difficulties in this section. For example, in architectures which rely on load-locked and store-conditional atomic primitives, any store operation can have the side-effect of clearing a reservation in another processor (regardless of silence), thus

impacting execution on the other processor. Furthermore, each dynamic store can have side-effects through the page table; most architectures specify some type of reference bits and/or dirty bits for each page to optimize page replacement policies. Suppressing update silent stores may cause a change in behavior if these side-effects are not accounted for properly at either the architecture or implementation level, potentially impacting program correctness.

In this section, we outline correctness issues known to us for many common ISAs in use as of the writing of this thesis. This list should not be considered exhaustive, but rather, an indication of complications that may arise. We ignore the compatibility argument made in other work [49] which is an important practical consideration in optimizing memory system design for industrial practitioners. We assume that allowing changed execution semantics is acceptable, provided the execution still adheres to the architectural definition.

We note that, in general, most existing ISAs are ambiguous where the subject of value locality is concerned, since they were designed before value locality was discovered in 1996 [73, 81]. In most cases, if we interpret “store” or “store-like” statements in the architecture to only imply changing the architected value, no substantial complication exists. Therefore, many of the scenarios described subsequently can be unambiguously resolved by a simple clarification to the ISA. However, we still explore various scenarios of interest to illuminate the possible issues.

4.5.2.1 PowerPC

We are most familiar with the PowerPC architecture, as that is the architecture used throughout the simulation environments of this thesis. PowerPC suffers from both

potential caveats mentioned previously, thus suppressing update silent stores may be observable through both load-locked (lwarx/ldarx) and store-conditional (stwcx/stdcx) atomic primitives, as well as through the page table reference and change (R/C) bits.

Focusing first on the page table, the reference bit is not a problem since store verification will still set this bit, if not set already, with the implementation we have described. According to the definition of PowerPC [78], setting this bit speculatively is permissible for store verifies carried out before a store is non-speculative. Regarding the change bit, this cannot be set speculatively with a store verification beyond an unresolved branch. However, if the change bit is already set, a speculative store verification presents no problem. Therefore, according to the architectural definition, speculative store verifies can be carried out with regular load operations provided the change bit is already set, or non-speculatively provided the store verify sets the change bit.

Load-locked (lwarx/ldarx) and store-conditional (stwcx/stdcx) present a more complicated scenario. In the following discussion we use lwarx to mean either lwarx or ldarx, and stwcx to mean either stwcx or stdcx for brevity. According to the architectural definition, any store to the reservation granule appearing between the lwarx/stwcx pair must clear the reservation, causing the stwcx to fail. If an update silent store occurs to the reservation granule, the reservation may not be cleared, thus causing a change in execution. Although the scenario is contrived, and we imagine no rational programming construct under which this may occur, it is possible to write code which behaves unexpectedly when update silent stores are suppressed. The conditions which must be met are represented graphically in Figure 4-8.

In order for an update silent store to be guaranteed to affect execution of

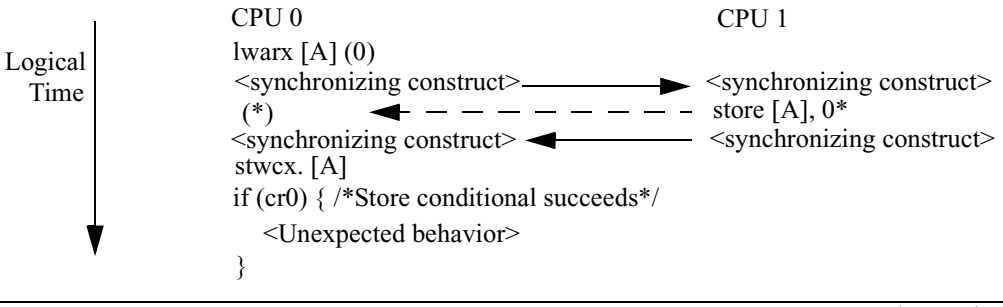

FIGURE 4-8. Conditions for Guaranteed Detection of an Update Silent Store. The update silent store is detected through its impact on load-locked/store-conditional behavior. The figure shows an update silent store to address [A] which can be detected through the success of the store-conditional performed by CPU 0 (through the value of the condition register cr0). The solid lines indicate causality constraint edges implemented by programming constructs surrounding the update silent store, which causes it to appear logically at (*) on CPU 0 (indicated by the dashed line). Once suppressing the update silent store affects execution of the stwcx, unexpected execution can occur.

lwarx/stwcx pairs, two conditions must hold: the update silent store must be constrained through program dependences to appear logically between the lwarx/stwcx, and the code must behave differently based solely on the success/failure of the stwcx and not the data value observed at the location of interest. The first condition ensures a simple logical reordering of memory references does not allow the update silent store to fall outside the lwarx/stwcx pair, the second condition ensures that the only architecturally visible aspect of the update silent store (its side-effect on the reservation) is exploited to change execution. Since PowerPC allows arbitrary memory references between lwarx/stwcx pairs,[7] it is indeed possible to create the described scenario. We illustrate two examples through PowerPC-like pseudo-code shown in Figure 4-9 and Figure 4-10. The first example shows code intended to implement mutual exclusion which works in some, but not all, cases when update silent store suppression is implemented. The second example shows how to detect whether update silent stores are being suppressed by exploiting the behavior of lwarx/stwcx.

7. Certain reference patterns are not advised and may lead to livelock if the size of the reservation granule is not considered [78].

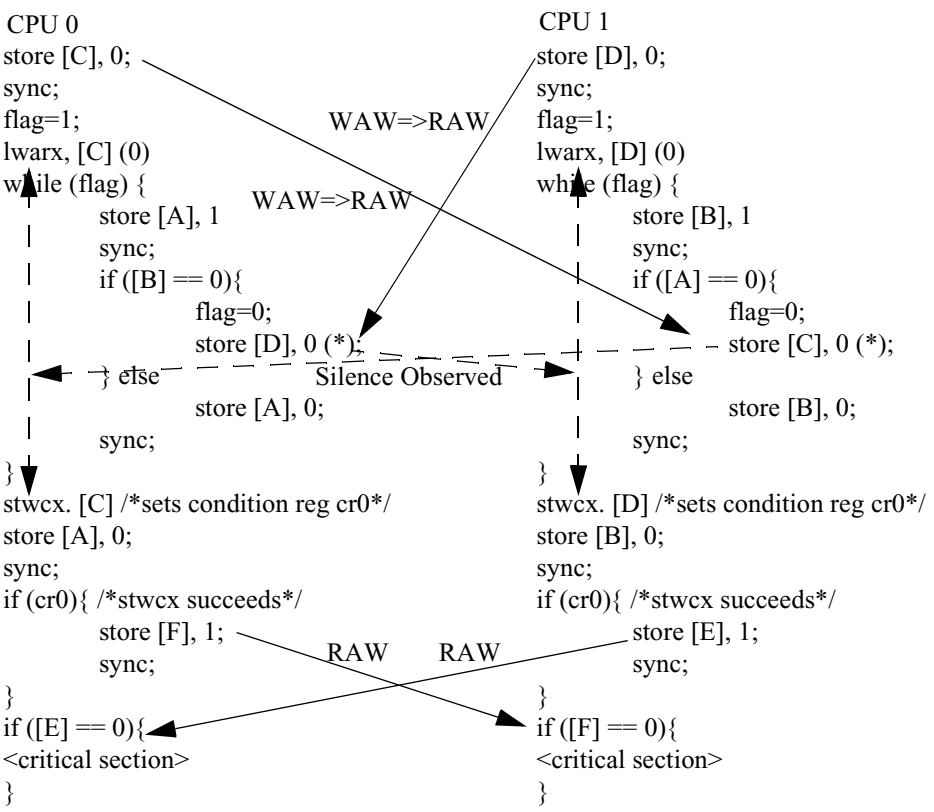

FIGURE 4-9. A Code Sequence No Longer Providing Mutual Exclusion. With update silent store suppression, the sequence shown may not longer provide mutual exclusion. The update silent stores which can be observed and affect program execution are shown (*). The same high-level constructs illustrated in Figure 4-7 are present and outlined.

Figure 4-9 shows synchronization code which may fail with update silent store suppression.[8] First, we argue that the construct shown provides mutual exclusion for the critical section labeled without update silent store suppression. The only case of interest is when CPU 0 and CPU 1 enter the sections protected by the lwarx/stwcx pairs at approximately the same time. Dekker's algorithm [98] guarantees that only the store to [D] by CPU 0 or the store to [C] by CPU 1 will be performed between the lwarx/stwcx pair. Let's assume without loss of generality that CPU 1 wins and its store to [C] occurs. In the case without update silent store suppression, the indicated edge will be a WAW, requiring an

8. The example is complicated and contrived. I apologize for that, but I couldn't think of a simpler one...

upgrade, clearing the reservation set by CPU 0; this condition is observed through the condition register *cr0* which is set to indicate success or failure of the stwcx. Therefore, CPU 0 will not enter the critical section (because its stwcx will fail), while CPU 1 will enter the critical section (because its stwcx will succeed), thus guaranteeing mutual exclusion. However, in the case of update silent store suppression, the update silent store edge is converted from a WAW into a RAW as described in Section 4.5.1. Therefore, an exclusive copy is not required, and the reservation set by CPU 0 (observed through condition register *cr0*) is not cleared. The stwcx on both CPU 0 and CPU 1 will succeed, allowing both to enter the critical section, which is not the intent.

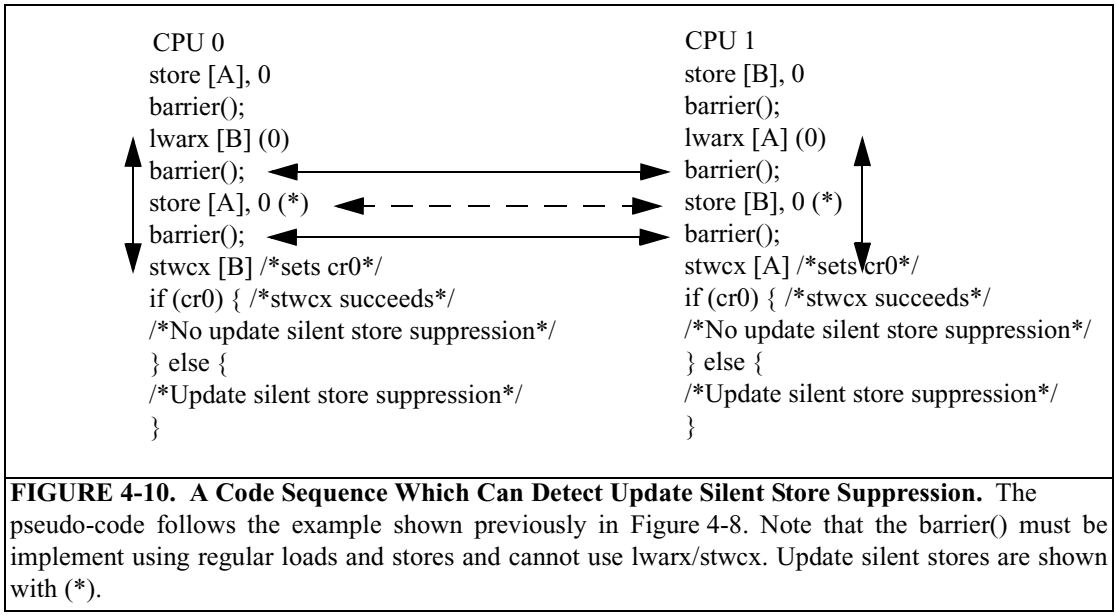

FIGURE 4-10. A Code Sequence Which Can Detect Update Silent Store Suppression. The pseudo-code follows the example shown previously in Figure 4-8. Note that the barrier() must be implement using regular loads and stores and cannot use lwarx/stwcx. Update silent stores are shown with (*).

Figure 4-10 re-illustrates the construct which can be used to verify whether update silent store suppression is being performed based on side-effectual communication through the reservation granule. The working of this example has already been explained in conjunction with Figure 4-8. Note that the barrier is any standard barrier-type synchronization [31] which must be implemented without using lwarx/stwcx and should not be to the same reservation granule as [A] or [B]. The additional barrier after each initializing

store serves to guarantee that both stores marked (*) are indeed update silent.

A final issue arises with store-conditionals (stwcx/stdcx) in PowerPC, again related to the reservation granule. If performing speculative update silent store suppression (as described in Section 4.5.1 where store verification is performed before a store reaches commit), we must guarantee that all stores (update silent or otherwise) clear any reservations set on the current processor if they are to the same reservation granule. We cannot clear the reservation speculatively when the store verify is issued because of potential livelock; the reservation register must be handled in program order. A simple solution is to check all stores (suppressed-update silent and non-update silent) against the reservation register at commit.

4.5.2.1 Other ISAs

Other common ISAs may exhibit characteristics similar to those described in detail for PowerPC. Again, the fundamental consideration is side-effects of memory references and architectural observability of these side-effects. We emphasize, even for PowerPC, most of the complication arises due to ambiguity of architectural definitions with respect to value locality. For example, while we showed that the code example of Figure 4-9 may not function as intended with update silent store suppression, a simple solution is to make such code illegal by disallowing architected side-effectual communication. Communication through the page table can be handled similarly; simply specifying that a change bit is not guaranteed to be set unless the value of the page has in fact changed is an intuitive and straight-forward solution.

For other load-locked/store-conditional architectures, e.g. MIPS [110] and Alpha [26], it may not be possible to detect update silent store suppression through side-effects

as we have illustrated. Alpha and MIPS do not recommend placing memory operations between load-locked/store-conditional pairs (the behavior of such constructs is either undefined or unpredictable), thus the problem indicated for PowerPC is eliminated.

In architectures which use other hardware synchronization constructs, e.g. test and set, compare and swap, fetch and add, the issue of side-effects is non-existent. Since the only visibility of memory operations is through their update to actual architected memory state, suppressing update silent stores should not lead to incorrect operation. Such architectures include x86 [25], AMD64/x86-64 [32], IA-64 [51], SPARC [63], and PA-RISC [47]. Finally, we consider the case of unaligned and multi-word memory reference instructions. In PowerPC and x86, atomicity of such references is normally not guaranteed, thus making silent store suppression correct; in other architectures, special consideration may be needed for multi-memory-word instructions.

4.6 Related Work

As far as we are aware, there has not been any published work exploring update silent stores, or similar concepts, in multiprocessor systems for the purpose of improving multithreaded program performance prior to this thesis research. Subsequent to the foundational research of this thesis, update silent stores have been explored by other researchers as a means to eliminate cross-thread dependences in both explicit threading and implicit threading environments. An example of explicit threading is thread-level speculation systems (TLS). Results in work by Steffan et al. [103] and Cintra et al. [24] showed that a substantial number of cross-thread memory dependence violations signaled solely because a location was written can be safely eliminated by suppressing update silent stores in such systems. An example of implicit threading is Slipstream processors. Update silent

stores have been utilized to in this environment to achieve similar benefit [104]. Other explicit and implicit threading architectures (a non-exhaustive list includes Multiscalar [39], dynamic multithreading or DMT [5], and master-slave speculative parallelization (MSSP) [113]) may also benefit from exploiting store update silence.

We have also discussed critical update silence and its interaction with coherence events to minimize system-level address communication. There has been substantial work on predicting and understanding multiprocessor sharing patterns which may be leveraged for update silence and critical update silence prediction (Section 4.4) [64] [59] [60] [86] [53] [74]. Many of these works study scientific workloads exclusively or assume directory-based coherence as a key enabler (Martin et al. [74] is a notable exception). Given our significant effort in commercial workloads and focus on snoop-based multiprocessors, we contribute significant validation and further engineering effort to these works. We discuss predictive coherence in the context of temporal silence in Section 5.7.

Chapter 5

Temporal Silence in Multiprocessors

We discussed update silent stores in multiprocessors in the previous chapter. In this chapter, we explore additional store value locality in multiprocessor systems to further reduce unnecessary value communication. Specifically, we now consider a form of temporal store value locality, which we call *temporal silence* or *temporally silent stores*. We defined the notion of temporal silence and temporally silent sharing (TSS) in Chapter 3—we will re-introduce it again shortly. We show the ability of TSS to eliminate unnecessary communication in multiprocessor systems and devise efficient methods to detect and communicate the occurrence of TSS in this chapter. We use the multiprocessor benchmarks from Table 3-4 and the simulation environment described in Section 3.6, unless stated otherwise, for all characterization data. We remind the reader that we discuss most mechanisms for capturing temporal silence from the perspective of snoop-based multiprocessor systems for the sake of brevity and clarity; comments on extending the proposals to directory-based schemes can be found in Section 5.8.3 and Section 5.9.

5.1 Motivation and Background

Both industrial practitioners and academic researchers alike have observed that communication-induced cache misses are a pressing performance limiter in modern systems, especially running commercial workloads. We have discussed these issues at length in Section 3.1, as well as the basics of invalidation-based cache coherence in Section 1.1. Furthermore, we have shown in Table 1-1 that a significant fraction of dynamic store operations are update silent and that update silent stores can be exploited to reduce both address and data traffic in multiprocessors throughout Chapter 4.

In this chapter, we explore *temporally silent stores* as a means to further improve communication performance in multiprocessor systems. To recap briefly (explained in detail in Section 3.3), temporal silence describes a program behavior in which we change a memory location to some intermediate value and a subsequent store reverts the location to an old value of interest. If this old value was previously present within the shared memory image, more specifically, in remote processor caches, we can imagine utilizing this to improve both communication latency and bandwidth within a multiprocessor system. We refer to the dynamic store writing the intermediate value as the *intermediate value store* and the dynamic store writing the old value of interest as the *temporally silent store*. The two stores together create a *temporally silent store pair.*[1]

We rigorously re-defined multiprocessor sharing to consider temporally silent stores leading to the definition of temporal silent sharing (TSS) in Section 3.3. Practically speaking, TSS can be thought of in this way: when a communication miss occurs, we determine whether the value returned by the memory system exactly matches the last value observed for the memory location by this processor before it was invalidated. If the values match, TSS has occurred. Throughout the rest of this chapter, when comparing the ability of temporal silence and update silence to eliminate communication misses, we model suppressing all update silent stores, i.e. USS-P (Section 4.2). We refer to this simply as USS in subsequent discussions.

5.2 Potential for Data Transfer Elimination

In Figure 5-1 we present infinite and finite cache miss rates for TSS as compared

1. Our discussion here describes the intermediate value store and temporally silent store as single dynamic stores of interest, presumed to be to the same memory location. In reality, we usually consider multiple locations, i.e. entire cache lines, which exhibit temporal silence; therefore, the intermediate value store and temporally silent store may not in fact be to the same location due to spatial/temporal interleaving of intermediate value stores and temporally silent stores.

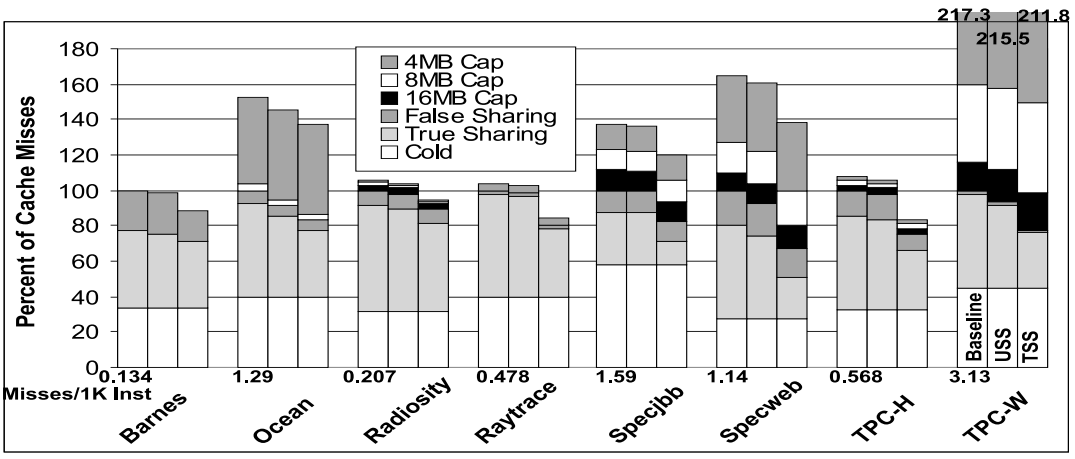

FIGURE 5-1. Percentage of Cache Misses for Different Definitions of Sharing. The data is normalized to the Baseline case for 64B cache lines and infinite caches. The contribution of additional capacity/conflict misses for progressively smaller, 8-way associative, finite caches of 16MB, 8MB, and 4MB is also indicated.

to USS, and also Dubois' definition [35], labeled "Baseline". The stacked bars (normalized to baseline, infinite cache) indicate the contribution of cold, true sharing, false sharing, and capacity/conflict misses for finite caches of decreasing sizes (16MB, 8MB, and 4MB). Focusing first on the infinite cache results, we see that TSS can reduce the overall miss rate for infinite caches by up to 33% and 27% over the baseline and USS, respectively (in *specweb*). The harmonic mean reduction across the scientific benchmarks (SPLASH-2) is 15% and 12%, respectively, and for the commercial workloads, 25% and 21%.

In the case of finite caches, the relative reduction in overall miss rate is smaller because of two effects: 1) Additional misses created by pure capacity and conflict misses; 2) Creation of additional capacity and conflict misses due to fewer invalidated lines, and therefore a larger working set (as explained for USS in Section 4.2). However, for all finite caches, we see the reduction in misses tracking the infinite cache reductions in absolute number of misses prevented. The only notable exception is in *tpc-w* for 4MB caches, where the reduction in overall miss rate drops from over 20% to less than 6% (normalized

to misses in the baseline, infinite cache case).

We can see in Figure 5-1 that the contribution of cold misses is substantial for many of the workloads, in large part due to the limited amount of system time we can practically simulate; we expect the relative fraction of cold misses to be substantially smaller in a real system. Because exploiting TSS in broadcast-based SMPs will not benefit cold misses, we focus solely on communication misses in subsequent exploration in this thesis.[2] To more clearly illustrate the impact of USS and TSS on communication misses, we break this component out separately in Figure 5-2.

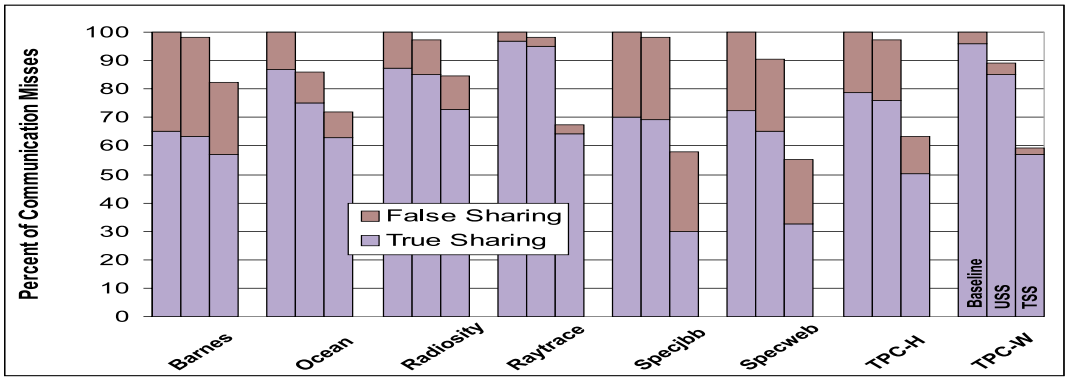

FIGURE 5-2. Percentage of Communication Misses for Different Definitions of Sharing. The data is normalized to the Baseline case for 64B cache lines and infinite caches.

Examining the communication miss component only in Figure 5-2, we see up to 45% reduction normalized to baseline and 35% reduction normalized to USS (in *specweb*). We see harmonic mean reductions in the scientific benchmarks of 24% and 19%, respectively; 42% and 32% for the commercial workloads. Most of the improvement is in true sharing, although some false sharing is also eliminated. Interestingly, TSS provides substantial benefit over USS, particularly in the commercial workloads. We will explore possible explanations for this throughout subsequent sections.

2. In directory-based systems, cold miss latency can be reduced for data which has been modified by another processor, converting 3-hop cold misses into 2-hop cold misses.

Finally, it is worthwhile to note the reduction in communication misses possible by exploiting store value locality for differing cache line sizes. Although we do not present detailed experimental data for brevity, we have observed that USS and TSS eliminate a smaller fraction of communication misses for increasing cache line sizes. This result is intuitive; as cache line size increases, the likelihood that an entire cache line exhibits silence to all written locations (leading to either USS or TSS avoidable misses) decreases. However, non-trivial opportunity for reducing communication misses can still be achieved for longer cache lines. Appropriate choice of cache line size is a function of many factors, and has been discussed in detail elsewhere [31].

5.3 System-Level Considerations for Effectively Exploiting Temporal Silence

The potential to reduce communication by exploiting temporal silence is large. However, exploiting temporal silence is inherently more difficult than update silence. To utilize temporal silence we must consider both dynamic stores (intermediate value store and temporally silent store) which comprise the temporally silent store pair; for update silence we need only consider a single dynamic store. This concept is explained in detail in Section 3.1 on page 57 and illustrated graphically in Figure 3-1. Because of the intermediate value, we decompose the problem of efficiently exploiting temporal silence into two subproblems: detection of temporal silence and communication of temporal silence. *Detection* is determining that a temporally silent store has occurred; *communication* is informing remote processors of its occurrence so they can avoid communication misses.

In order to detect temporal silence we must provide facility for storing both the intermediate value and the temporally silent value for a given memory location; the intermediate value is needed to maintain coherent memory state, the temporally silent value is

needed to detect reversion. Therefore, efficient detection implies minimizing storage for previous versions of the cache lines which are candidates for reversion. Efficient communication implies system transactions should only occur when misses can be eliminated, thus reducing communication overhead. In order to optimize detection and communication we may utilize the time which occurs between the temporally silent store and a remote access to the cache line. To make this idea more concrete, we present Figure 5-3.

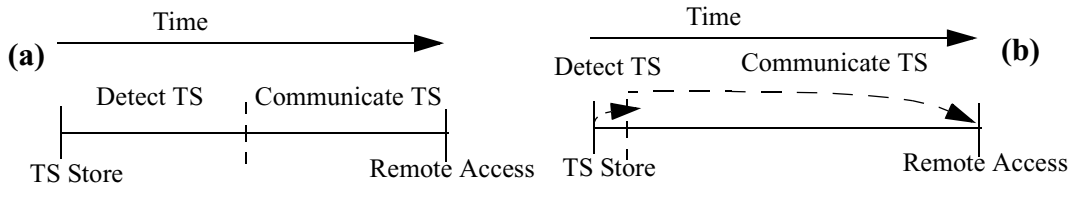

FIGURE 5-3. Illustration of Timeliness Aspects in Temporal Silence. When eliminating communication misses, we can utilize the time between the temporally silent store (TS Store) and a remote access to the written data to improve detection and communication efficiency. In part (a) of the figure, we indicate this time can be used to perform the two essential aspects, detection and communication. In part (b), we show that achieving system-level benefit makes maximizing the time used for communication desirable. The dashed line indicates we desire quick detection maximizing communication time.

In Figure 5-3(a), we show that the interval between the temporally silent store and a remote access may be used for both detection of temporal silence and communication of its occurrence to other processors. To effectively remove communication miss latency experienced by remote processors, we must ensure that they are informed of temporal silence before a remote access, otherwise potential benefit may be lost. Therefore, it is desirable to inform as quickly as possible since we do not know when a remote access will occur; Figure 5-3(b) depicts this goal. When considering different detection and communication schemes throughout subsequent sections, we must consider this timeline and the effect each detection and communication scheme has on it. Note that some of the schemes we explore may be able to perform either of these aspects (detection or communication) implicitly, that is, without an explicit comparison or transaction. If either aspect can be

performed implicitly, it need not be considered as part of the timeline. As an example, consider update silent stores; with the USS-Hit policy shown in Section 4.2 on page 76 communication is performed implicitly because the upgrade normally present is simply avoided. Therefore, the timeline collapses into the single aspect of detecting the store is update silent.[3]

In addition to the timeliness aspect, another key aspect to consider in efficient detection is the number of system-visible versions of a cache line which occur between TSS eliminated communication misses. Multiple versions can occur because of the definition of TSS, which only stipulates that the value returned by the memory system for a reference be the same as the last value the processor of interest observed. Obviously, this value can be unique for each processor in the system. We show a contrived example of such a scenario in Table 5-1.

Table 5-1: Code Example for TSS Indicating Multiple Intermediate Versions. Coherence transactions for a core with USS store suppression are shown (LD [A] at T0 returns 0). The LD misses at T6, T9, and T12 can be eliminated with ideal TSS, but multiple system-visible versions must be tracked for all TSS misses to be successfully avoided.

	CPU 0		CPU 1		CPU 2	
Time	**Instruction**	**Cmd/Txn**	**Instruction**	**Cmd/Txn**	**Instruction**	**Cmd/Txn**
T0	LD [A] (0)	Read/Miss	LD [A] (0)	Read/Miss	LD [A] (0)	Read/Miss
T1			ST [A], 1	Invalidate		
T2	MEMBAR		MEMBAR			
T3	LD [A] (1)	Read/Miss				
T4			ST [A], 0	Invalidate		
T5			MEMBAR		MEMBAR	
T6					LD [A] (0)	Read/Miss
T7			ST [A], 1	Invalidate		
T8	MEMBAR		MEMBAR			
T9	LD [A] (1)	Read/Miss				
T10	ST [A], 0	Invalidate				
T11	MEMBAR				MEMBAR	
T12					LD [A] (0)	Read/Miss

3. We will take advantage of viewing update silent stores as a special case of temporally silent stores, and their interaction with this timeline, again in Section 5.7.4.

In this example, Read/Misses at T6, T9, and T12 are all TSS, as each returns the value previously observed for address A by CPU 0, 2, and 1, respectively. However, we see that at least two versions of the value at address A must be tracked in order for all misses to be prevented (both 0 and 1), as the live ranges [21] for these uses of address A overlap. The live ranges for both values are indicated in the table with curved arcs.

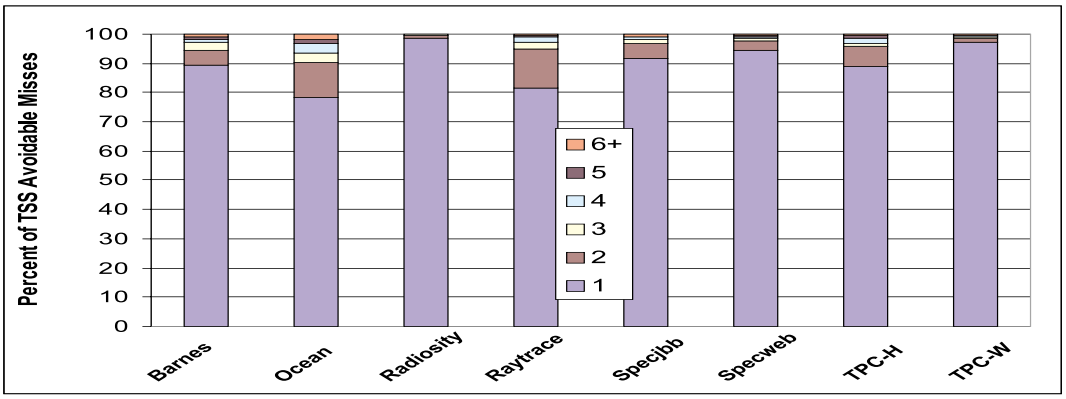

FIGURE 5-4. Globally Visible Cache Line Versions Required for Exploiting Useful TSS. The percentage of TSS avoidable misses captured with increasing number of globally visible versions is indicated. A globally visible version is created on any remote request to a modified (M) line. Globally visible version distance zero constitutes USS, and is not shown for clarity.

To obtain a first-order understanding of how many versions are actually required in practice, we perform a simple study. We count, for each occurrence of TSS which prevents a remote miss, how many globally visible versions of the cache line have been transferred among processors in the system between the original invalidation of the cache line and the TSS avoidable miss. Each remote request to a modified (M) line creates another globally visible version.[4] The results of this study are shown in Figure 5-4.

We can see from the figure that a version distance of one is sufficient for the four-

4. Note that this definition does not exactly correspond to the live ranges we discussed previously in our example because it does not consider multiple versions which have the same value contained within the cache line. However, this metric does provide a meaningful measure of the distance in system-visible versions which must be captured. We use this metric instead of live ranges because it corresponds more directly to implementation within a coherence protocol framework; this point will become more clear when we discuss coherence protocol enhancements in Section 5.4.2.

processor system we study, capturing over 90% of the opportunity in all cases (except *ocean* and *raytrace*). Therefore, when exploring methods of efficient detection and communication throughout subsequent sections, we focus on capturing only a distance of one. We refer to this single candidate for reversion as the *previous version*, with the understanding that it is in fact the previous system visible version. When appropriate, we discuss the ability of different methods to capture additional previous versions.

5.4 Communicating Temporal Silence

Having illuminated that there is significant potential to exploit temporal silence in multiprocessor systems to eliminate communication misses (Figure 5-2), and illustrated system-level aspects to consider in exploiting temporal silence, we first turn to the subproblem of communication. We explore communication first because it is likely to be the most difficult aspect to implement; correctly reasoning about and validating memory ordering and coherence protocols is widely regarded as a difficult problem [75] [1]. Once we understand the trade-offs of each communication mechanism, we can then design efficient detection suited to each. This flow is sensible since detection is conceptually straight-forward; we need only provide adequate intermediate value and temporally silent value storage.

Throughout subsequent sections we discuss both speculative and non-speculative mechanisms which enable communication of temporal silence. Since these terms are widely used with varying interpretation in computer architecture, we feel a description of their intended meaning in this context is appropriate. We use the term *speculative* to refer to any mechanism which fundamentally requires the ability to recover a previous architected machine state to extract benefit from temporal silence. We use the term *non-specu-*

lative to refer to any mechanism which does not require this ability. Note that both speculative and non-speculative communication mechanisms can be employed in machines with pre-existing support for other forms of speculation, e.g. branch prediction. We provide additional discussion of each mechanism throughout the following sections.

5.4.1 WC TSS

Many proposals, e.g. Dubois et al. [34], exploit the freedoms granted under weak memory models to optimize shared memory performance through either delaying processing of remote invalidations or sending of invalidations. These proposals expose the architected behavior of the consistency model to the coherence protocol allowing relaxed read/write timing. Similarly, we can utilize weak consistency models to exploit some cases of TSS. Weak consistency models, in their definition, allow some store pairs to be collapsed into an atomic unit non-speculatively (Figure 3-1). In PowerPC weak ordering, temporally silent pairs without a memory barrier, i.e. sync instruction, between them can be collapsed into an atomic unit, avoiding an invalidation, and preventing some TSS misses. We call our scheme which eliminates these misses weak consistency TSS (WC TSS). However, if a memory barrier divides the temporally silent pair, the pair cannot be collapsed by exploiting weak ordering semantics and a more aggressive method must be explored.

In Figure 5-5 we show the reduction in communication misses possible using an infinite write buffer and ordering intermediate values only at memory barriers, which indicates the best possible performance for WC TSS for the observed execution. The format of the figure is the same as Figure 5-2, except we have added the category “WC TSS” to indicate temporal silence captured exploiting weak consistency. We see relatively small

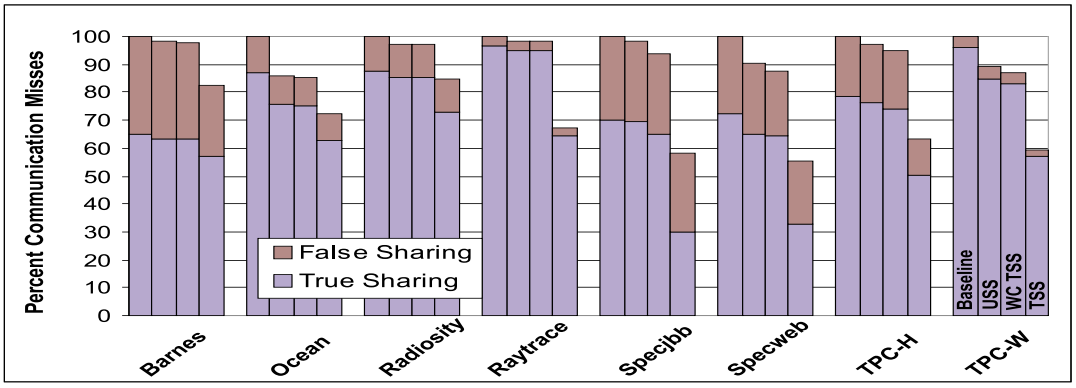

FIGURE 5-5. Percentage of Communication Misses for WC TSS. The data is normalized to the Baseline case for 64B cache lines, including the reduction in communication misses under both USS and TSS, as well as exploiting weak memory ordering semantics to collapse atomic temporally silent pairs.

reductions in communication misses beyond USS (a maximum of 3% in *specjbb*), indicating that simply exploiting weak memory models is not sufficient. Considering the timeline presented in Section 5.3, this technique utilizes implicit communication, therefore the timeline collapses to simple detection. As long as temporal silence is detected before a resource or memory ordering constraint occurs, the temporally silent pair is communicated by simply removing the invalidation request.

5.4.2 The MESTI Coherence Protocol

In machines with consistency models that require all stores to be ordered, WC TSS will be impossible without speculation [93]. Additionally, we saw in the previous section that exploiting weak memory models does not capture a significant fraction of the benefit possible with TSS, indicating that many temporally silent pairs cross memory barriers. As an example, consider lock variables. Locks are splendid candidates to exhibit temporal silence since they revert to the unheld value when released, but must be protected by memory barriers in order to implement their essential function.

We propose augmenting the coherence protocol as a non-speculative means to

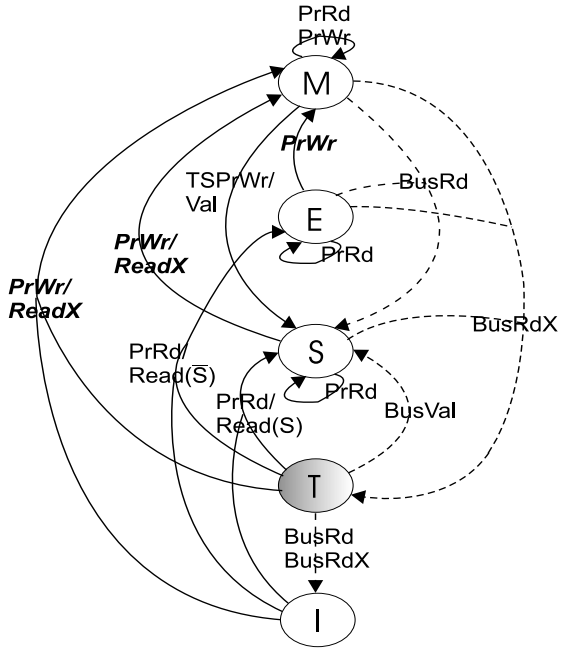

FIGURE 5-6. State Machine for the MESTI Protocol. We augment MESI (using the notation from Culler and Singh [31]) with temporal silence support by adding a *temporally invalid* (T) state. *Val* denotes the validate transaction used to communicate the occurrence of temporal silence against the previous value for a cache line. The italicized and boldface PrWrs indicate where a previous version is saved.

exploit temporal silence in cases where the intermediate value store must be ordered. In Figure 5-6, we show the additional coherence support we propose. We call our proposed protocol *MESTI*, which adds the temporally invalid state *T*. This state is entered upon receipt of an invalidate from another processor in the system if the data was previously valid (M, E, or S states). If another bus transaction occurs to a line in T state, it transitions to I state, unless the transaction is a validate, in which case it transitions to S state. Entering T state allows remote processors to save the previous version of a cache line so it can be reverted to later. The validate transaction allows a remote processor to re-install a cache line in remote caches when temporal silence is detected. In Figure 5-6, we indicate the PrWrs in boldface and italic text when a previous version of the cache line is saved; this enables a subsequent validate transaction if the line reverts to this version.[5]

A validate effectively places a cache line back into remote caches with simply an address transaction, as opposed to sending new data, as is done in update protocols. However, address traffic may not be equal to an update protocol because a validate is triggered only when the final temporally silent store occurs to the cache line; multiple intervening stores not part of the temporally silent pair or other temporally silent stores to different locations within the cache line do not cause validates.

We provide discussion of validation of the MESTI protocol in our performance simulation environment Section A.1, as many of the issues uncovered in implementation are specific to PowerPC and the base coherence implementation used. A detailed description of the protocol used in the performance simulator can be found in Section A.2. Qualitatively, the protocol change we propose is inherently simpler than many other protocol changes because T state is used for performance optimization and is handled similarly to I state during race conditions to ensure correctness. The principal difficulty in designing a correct implementation of MESTI is to ensure that a validate does not become associated with incorrect T state cache lines. There are many methods to ensure this; we will discuss the necessary conditions for failure of the protocol as well as widely-applicable solutions in detail in Section 5.8.3.

5.4.2.1 Temporal Silence Captured

In Figure 5-7, we show the reduction in communication data traffic possible with MESTI. We see harmonic mean reductions in the scientific benchmarks of 21% and 15% over the baseline and USS cases; and 40% and 32% for the commercial workloads. Note

5. We assume a write-allocate cache to save the previous globally visible version for the I to M and T to M transitions. Note that the version saved here is the data arriving from the system before the line is modified. Furthermore, this diagram assumes an “implicit writeback” upon any modified intervention, i.e. any remote request to a modified line also writes the modified data back to memory. This is standard handling in the Illinois MESI protocol [31].

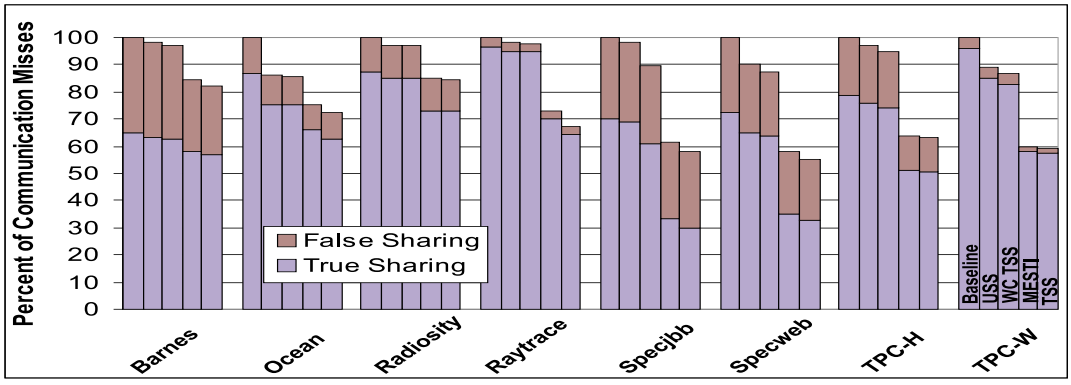

FIGURE 5-7. Percentage of Communication Misses for MESTI. The data is normalized to the Baseline case for 64B cache lines, including the reduction in communication misses under USS, WC TSS, and TSS for comparison purposes.

that MESTI is within 4% of the TSS limit in communication misses for all workloads except *raytrace* (6%). Also note that the relative ability of MESTI to approach the TSS limit is slightly larger in the commercial workloads versus the scientific workloads, indicating that available opportunities for capturing temporal silence are different between workload types. We will explore this comparison between workload types more extensively in Section 5.10.

5.4.2.2 Writeback Elimination

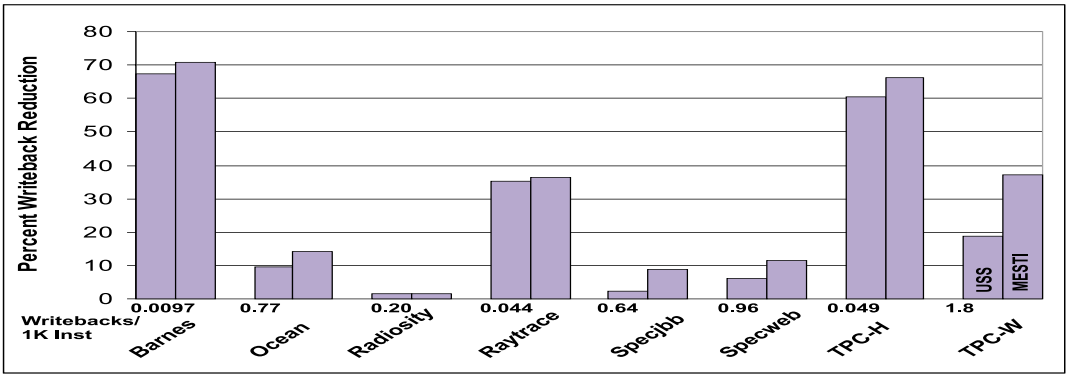

FIGURE 5-8. Writebacks Removed by Exploiting Temporal Silence with MESTI. The data is normalized to the Baseline case for 64B cache lines with an 8-way associative, 4MB cache. Results for USS are also shown for comparison.

In addition to communication misses, we can also exploit TSS to eliminate cache writebacks, as shown for USS in Section 4.2. Since MESTI only captures cases of TSS

which correspond to the previous version of a cache line, we can also directly eliminate writebacks under this protocol.[6] We show writebacks eliminated under MESTI for a 4MB, 8-way associative cache in Figure 5-8. We see that in most cases, the explicit writeback reduction possible through MESTI is modest beyond USS. However, these results only indicate writeback reduction due to explicit modified line cast outs. As discussed in Section 5.4.2, any dirty miss under MESI creates an implicit writeback. Therefore, communication misses eliminated with MESTI will also reduce implicit writebacks. In MOESI protocols, used for performance simulation in Chapter 6, this implicit writeback is not needed due to the O (owned) state.

5.4.2.3 MESTI and TSS Compared

Table 5-2: Code Example for MESTI. Coherence transactions for a core with USS store suppression and weak ordering are shown (LD [A] at T0 returns 0). The LD miss at T5 can be eliminated with MESTI.

	CPU 0		CPU 1	
Time	**Instruction**	**Cmd/Txn**	**Instruction**	**Cmd/Txn**
T0	LD [A] (0)	Read/Miss	LD [A] (0)	Read/Miss
T1	ST [A], 1	Invalidate		
T2	MEMBAR			
T3	ST [A], 0	Validate		
T4	MEMBAR		MEMBAR	
T5			LD [A] (0)	*Read/Miss

To illustrate why MESTI is incapable of exploiting all cases of TSS, we show two sample load/store sequences, one that can be exploited (in Table 5-2) and one that cannot (shown previously in Table 5-1, duplicated here in Table 5-3). In Table 5-2, the Read/Miss at T5 can be eliminated because the data it requires is validated at T3. However, Table 5-3 shows a more complicated scenario in which the load misses at T6, T9, and T12 are all TSS because the values read do match the value seen previously by each CPU, but are not

6. Eliminating writebacks for general TSS is not straight-forward because of multiple intermediate values and temporally silent values, as discussed in Section 5.3.

captured with MESTI. In short, MESTI is not able to exploit all cases of TSS because it only allows reversion to the immediately previous version of a cache line. Depending on sharing patterns, multiple versions which differ between CPUs may be required, as was also shown in Figure 5-4. However, the relatively simple, non-speculative, protocol seems promising.

Table 5-3: Code Example for TSS Indicating Multiple Intermediate Versions. Coherence transactions for a core with USS store suppression are shown (LD [A] at T0 returns 0). The LD misses at T6, T9, and T12 can be eliminated with ideal TSS, but multiple versions must be tracked for all TSS misses to be successfully avoided.

	CPU 0		CPU 1		CPU 2	
Time	**Instruction**	**Cmd/Txn**	**Instruction**	**Cmd/Txn**	**Instruction**	**Cmd/Txn**
T0	LD [A] (0)	Read/Miss	LD [A] (0)	Read/Miss	LD [A] (0)	Read/Miss
T1			ST [A], 1	Invalidate		
T2	MEMBAR		MEMBAR			
T3	LD [A] (1)	Read/Miss				
T4			ST [A], 0	Invalidate		
T5			MEMBAR		MEMBAR	
T6					LD [A] (0)	Read/Miss
T7			ST [A], 1	Invalidate		
T8	MEMBAR		MEMBAR			
T9	LD [A] (1)	Read/Miss				
T10	ST [A], 0	Invalidate				
T11	MEMBAR				MEMBAR	
T12					LD [A] (0)	Read/Miss

The results shown in Figure 5-7 assume enough stale storage throughout the processor core and memory hierarchy to detect all cases of TSS which MESTI can exploit. Principally, this means augmenting traditional cache structures with stale storage of 64B per cache line, matching the cache line length. This implies a doubling of the data storage capacity of the cache. Also, as soon as a cache line has become temporally silent due to a temporally silent store, we broadcast a validate transaction to all other processors in the system. We will explore more efficient methods of implementing stale storage for a single previous version as needed by MESTI in Section 5.5. We will discuss delayed validate

broadcasts and other address traffic considerations in Section 5.7.

Considering the timeline presented in Section 5.3, MESTI assumes explicit communication; an invalidate for the intermediate value plus a validate when temporal silence occurs. It also assumes explicit detection against the system-visible value. Therefore, we must consider both aspects in designing a system using MESTI. As mentioned, we will discuss both aspects in detail in Section 5.7 and Section 5.5, respectively.

5.4.3 Speculative Lock Elision (SLE)

As explained in Section 3.1, key problems in exploiting TSS are maintaining correct system operation considering the intermediate value, but also preventing communication misses when temporal silence occurs. WC TSS creates atomic temporally silent pairs through exploiting relaxed memory consistency constraints; MESTI allows both the intermediate value and temporally silent store to be visible through augmented coherence support. Note that both methods are completely non-speculative in nature and may also be implemented entirely outside the processor core.

Another method to create atomic temporally silent pairs is through speculation. Rajwar and Goodman propose Speculative Lock Elision (SLE) [93] which exploits atomic "silent store-pairs" to provide a mechanism for correctly executing critical sections with non-conflicting data accesses in parallel. SLE also avoids the lock transfer which eliminates remote misses on the lock variable. This approach enables creation of atomic temporally silent pairs regardless of the architected consistency model [93].

Since this approach is speculative in nature, all standard barriers to speculation apply, i.e. non-cacheable references and I/O accesses. However, changes in program flow, either due to microarchitectural state, e.g. branch predictions, or exceptions/interrupts do

not require the speculation to fail.[7] Also, this approach requires sufficient buffering and roll-back support within the processor core for speculative operations and speculative memory writes. Any lack of resource within the processor or memory system may cause the elision process to fail. Various approaches for providing adequate resources for speculation are discussed by Rajwar [93] [90]. In comparison with WC TSS, creating atomic temporally silent pairs through speculation creates additional complications within the memory system to enable performing multiple memory writes as an atomic unit.

Because SLE allows non-conflicting critical sections to be executed concurrently, we cannot study its performance, strictly speaking, with trace-based analysis. The basic problem stems from SLE's implicit reordering of non-conflicting critical section executions. Given a fixed trace the executions cannot be reordered, they are constrained by the trace; if multiple processors attempt to enter the same critical section, the base simulator creating the trace serializes their executions within the critical section regardless of conflicts. We emphasize that this is not a problem with SLE, it is a difficulty in using trace-based analysis to quantify its performance potential. Therefore, we refrain from attempting any kind of miss-rate analysis for SLE in this section and discuss detailed performance evaluation in Section 6.6.5. The inaccuracy of trace-based analysis occurs in the case of contended critical sections, where SLE can expose additional cases of TSS not reflected in our trace-based analysis. We also note that MESTI cannot capture and exploit such cases in its nature, which is a fundamental difference between the two approaches. However, in the case of uncontended critical sections, SLE and MESTI can capture the same set of TSS

7. However, in practice, interrupt and exception conditions will likely cause a sufficient number of instructions to occur between the "silent store-pair" such that elision will fail due to resource constraints. This is acceptable, as the elision can always be aborted.

avoidable misses for all idioms detected by SLE; MESTI may exploit additional cases of TSS which cannot be collapsed into an atomic region or which avoid SLE's idiom detection.

In addition, SLE and MESTI are fundamentally different in their approaches to exploiting temporal silence. Since SLE relies on speculatively creating atomic regions, it must know when to ***start*** speculating, i.e, which store is the intermediate value store. Naively, any store is a candidate to begin the elision process. Attempting to elide every store has the principle shortcoming of leading to many false positives for elision candidates. A simple way to reduce the candidate set of intermediate value stores is to detect certain idioms [93] as high-confidence for elision. In that work, they treated only store-conditional operations as candidates for elision; more specifically, a pattern of load-locked/store-conditional operations to the same memory address, followed by a subsequent store. This works well for the studies presented there, since only user code is simulated for scientific benchmarks from the SPLASH-2 suite [107] where all synchronization code is supplied by the user. Since store-conditionals were only used for lock acquires, this essentially provides perfect instrumentation for their targeted references around critical sections. We have observed, in full-system simulation environments under AIX/PowerPC, that the load-locked/store-conditional part of the idiom which can begin speculation is much more prevalent. It is used in additional contexts which are normally not good candidates for elision: clearing the reservation on context switches, to implement atomic list insertion/deletion, to perform lock releases in addition to acquires, etc. Therefore, a stricter idiom or instrumentation may be required in practice to obtain good performance.

Another possible method to begin the elision process not based on a particular

instruction idiom, and not explored here, is to consider any write which modifies a cache line from the previous version as a candidate to begin the elision process. Practically, this is any pure store miss or store upgrade. However, such an approach is likely much more difficult to integrate into methods of speculation recovery support, since this attribute is not determined until execution of the store; most existing recovery mechanisms rely on program order to ease implementation, e.g. dispatch/insert of the store as is assumed by Rajwar et al. [93]. For this approach, determining the precise recovery state may prove difficult for the register checkpoint approach, leaving only buffering within the core as potentially feasible.

In contrast, MESTI is a purely non-speculative, reactive approach; it knows which value is the temporally silent value and simply detects when reversion to this value occurs. MESTI can be completely implemented outside the processor core with no speculation support. MESTI does not rely on detecting any particular idiom in order to facilitate eliminating communication misses. However, when MESTI and SLE are both eliminating the same communication misses, SLE can do so at lower cost in terms of address traffic. MESTI requires explicit communication through upgrade/validate pairs to eliminate communication misses. SLE utilizes implicit communication by allowing caches to continue using previous copies of the cache line.

As a final note of comparison for SLE and MESTI, we consider the stability of both methods in the face of data alignment and aggressive coherence implementations. Researchers have proposed collocating lock variables and critical section data to avoid multiple cache misses to shared data; one miss for the lock itself and others to the data within the critical section itself [31]. This may be beneficial for uncontended, fine-

grained, locking scenarios when acquiring a specific lock is a reliable predictor of data which will be accessed as part of the critical section. If collocated data is written, the elision process will fail, as writes to variables other than the lock variable itself are not elided and require exclusive ownership. Under MESTI, the protocol need not fail by design, as it is agnostic to collocation issues. If collocated data exhibits temporal silence, MESTI can still achieve performance benefit; whether collocated data exhibits temporal silence is algorithm-specific.

Furthermore, aggressive coherence implementations which speculatively prefetch exclusive ownership of cache blocks to reduce store commit latency for strong memory models, e.g. [25], [30], [33], or reduce read/upgrade pairs for migratory data, e.g. Kaxiras et al. [53], as well as software exclusive prefetches, may interact with both mechanisms. Under both SLE and MESTI, useless exclusive prefetches to the lock variable will cause both algorithms to restart or fail. However, exclusive prefetches to data within the temporally silent pair create additional complication for SLE. Since SLE uses the coherence protocol to detect atomicity violations, any exclusive prefetch which hits data between the temporally silent pair will be considered an atomicity violation. Therefore, it will trigger a restart or failure, leading to performance degradation. This effect should become more pronounced as the temporally silent pair distance (program distance between intermediate value and temporally silent store) lengthens, practically limiting the temporally silent pair distance SLE can exploit. In contrast, because MESTI does not rely on speculatively creating atomic regions using the coherence protocol, its performance will be less affected by such artifacts of modern implementations and other aggressive coherence schemes. We show in Section 5.5.3 that temporally silent pair distance can be substantial, indicating that

providing robust mechanisms immune to other coherence activity to exploit long-distance temporally silent pairs is worthwhile. Finally, note that the same phenomenon can occur if the amount of false sharing to data within the speculative critical section is substantial due to data alignment issues.

We show a detailed performance comparison of SLE and MESTI in Section 6.5 and Section 6.6.5. We also describe our implementation of speculation support for both register and memory buffering in that section after the detailed simulation environment for performance studies is discussed.

Considering the timeline presented in Section 5.3, SLE assumes implicit communication, but relies on explicit detection of temporal silence to determine when the elision is successful; the elided intermediate value store must be paired with a temporally silent store to be successful. This is accomplished by Rajwar et al. [93] through idiom detection and explicit storage for elision candidates as well as both speculative execution and speculative write retirement support. Therefore detection is low cost and timely, as long as proper idioms are used.

5.4.4 Temporal Silent Sharing (TSS)

Although we showed in Figure 5-7 that MESTI can capture many TSS misses, other workloads may be less amenable. We have shown an example of code which exhibits TSS but is not exploitable via MESTI in Table 5-3. Furthermore, changing the coherence protocol, as required for MESTI, or adding SLE support to the processor core may be undesirable.

Seminal research by Lipasti and Shen [73] introduced value locality and value prediction to the academic community and promoted load value prediction (LVP) as a poten-

tial application of value locality. Since this thesis builds on the foundation of value locality, it is fitting that we can return to this original work to provide another avenue to exploit TSS. Until now, we have focused on the value locality of the stores themselves which is memory value production; however, the definition of TSS leads to another simple speculative method to potentially capture the benefit.

In Section 5.1, we discussed that TSS occurs when the value returned from the memory system on a miss is the same as the previous value observed by that processor. Given the large fraction of TSS communication misses, using values from tag-match invalid cache lines may prove fruitful. This approach has the desirable property of capturing all TSS misses, additionally all TSS false sharing misses, and a subset of TSS true sharing misses[8]. However, this method has to address the challenges of implementing correct load value prediction in multiprocessors [76] and necessitates actual data transfer for the predictions to be verified. The previously described methods avoid data transfer when TSS has occurred. This implies data traffic for this method will be higher than WC TSS, MESTI, or SLE. We discuss our method of mis-speculation recovery and performance of this approach, in Section 6.6.3. Since detection and communication of temporal silence are implicit from the producer's perspective with this approach, the timeline presented in Section 5.3 does not apply.

5.5 Detecting Temporal Silence

Now that we have described methods at our disposal to communicate temporal silence, we turn our focus to detection. As described in Section 5.3, our goal is to detect

8. The subset of interest includes those cache lines which are truly shared, but there is sufficient time between a falsely shared demand miss and the eventual access which causes true sharing. The demand miss will be value verified to be correct and will have prefetched for true sharing.

temporal silence as quickly as possible. However, slight delay in detection may reduce its cost. It is also important to note that any detection technique, in addition to being space efficient, must be capable of capturing the dynamic program distance between the temporally silent pair (as described in Section 3.1); otherwise, temporal silence will not be observed and some opportunity to improve communication performance may be lost. We explore low cost detection techniques considering both aspects throughout this section.

5.5.1 In the Processor Core

If we assume a processor that implements update silent store suppression, we can augment the load/store queue (LSQ) to keep the load data for a verified store in the LSQ. This is similar to the LSQ cache presented in Section 2.3.2. Then detecting temporal silence simply involves checking any new store data values against the previous value for that location, i.e. the oldest loaded value for that address in the LSQ. Any stores which become temporally silent in the queue can be dropped, while all other stores are performed as usual.

Store buffers or write caches [83] [63], can also be modified to perform a similar function. If the store buffers are write allocate, e.g. UltraSparc-III [63], update silent store suppression need not be performed to detect temporal silence. Instead, explicit stale storage can record the memory value when the store buffer is allocated, before the store is actually performed. New stores are combined into the store buffer, as normal, and temporal silence can be detected by comparing the store buffer data against the stale version saved when the store buffer was allocated. If these match, the cumulative effect of all writes has resulted in temporal silence. Methods described throughout Section 5.4 can then be used to communicate the fact that temporal silence has occurred to other proces-

sors in the system. Note that these methods can detect temporal silence with very little delay, leaving maximal time for communication to remote processors (as described in Section 5.3). However, detecting temporal silence solely within the processor core or small write buffers or write caches can only capture temporally silent pairs with a short dynamic program distance between them. We will show in Section 5.5.3 that under many circumstances, particularly in commercial workloads, a long program distance can exist between useful temporally silent pairs. Thus, we focus on methods implementable outside the processor core to effectively capture this distance. Many of the techniques described subsequently are also applicable to microarchitectures with either write buffers or write caches.

5.5.2 Limited Stale Storage Throughout the Memory Hierarchy

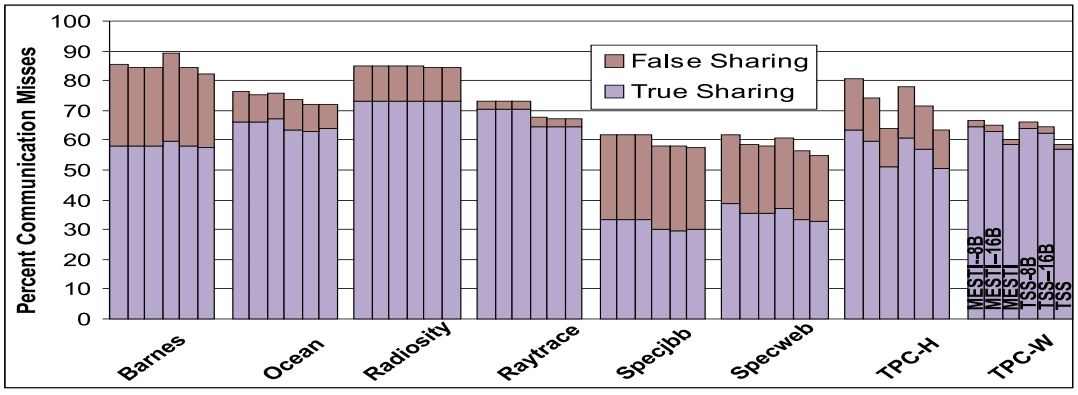

FIGURE 5-9. Effect of Limiting Stale Storage on Exploiting Temporal Silence. The reduction in communication misses normalized to the Baseline case (see Figure 5-7) for 64B cache lines and infinite caches for MESTI and TSS with limited stale storage (16B/cache line, 8B/cache line) is shown.

Stale storage of an entire cache line allows the best possible performance of MESTI, and is what we have shown in all results presented so far. However, this implies a doubling of cache storage in order to completely capture all cases of temporal silence. In Figure 5-9, we show the effect of limiting stale storage to a subset of an entire cache line—either two separate 8 byte blocks (16B case) or one 8 byte block (8B case)—for both MESTI and TSS[9]. We see that for all benchmarks, limiting stale storage to 16B (1/4

of a cache line) achieves nearly the ideal reduction in communication misses versus full cache line stale storage. The only exceptions are in *tpc-h* and *tpc-w*, where the difference is 5-10%. When stale storage is limited to 8B (1/8 of a cache line), *barnes* and *specweb* also show a non-trivial lost opportunity of 8% and 5%, respectively. However, in all cases, the reduction in sharing misses is substantial even with limited stale storage. Since most of the reduction with TSS is in true sharing misses, this implies that for sharing patterns which exhibit temporal silence, relatively few locations within the cache line are participating in the sharing. This observation can be exploited in many of the mechanisms we describe subsequently to limit stale storage per cache line, reducing the cost of detection substantially.

We also found that for 16MB caches, using nearly equivalent overall storage (a 16MB conventional cache vs. a 14MB, 7-way associative cache with ~1.8MB stale storage organized as 8B/cache line), the overall miss rate of the 14MB cache with either MESTI or TSS was better than a conventional 16MB cache with USS store suppression, further indicating that exploiting temporal silence may be an effective way to eliminate cache misses and improve performance. Judiciously limiting stale storage does not directly affect the timeline for efficient exploitation presented in Section 5.3, it only affects which TSS misses can be detected. We will further reduce the cost of detection throughout subsequent sections.

5.5.3 Temporally Silent Pair Distance—The Key to Efficient Stale Storage

Along with strictly limiting the amount of spatial locality tracked to detect tempo-

9. Limiting stale storage for general TSS does not correspond directly to an implementation in some sense (consider the example in Table 5-1). However, we still present characterizing data for this case for completeness.

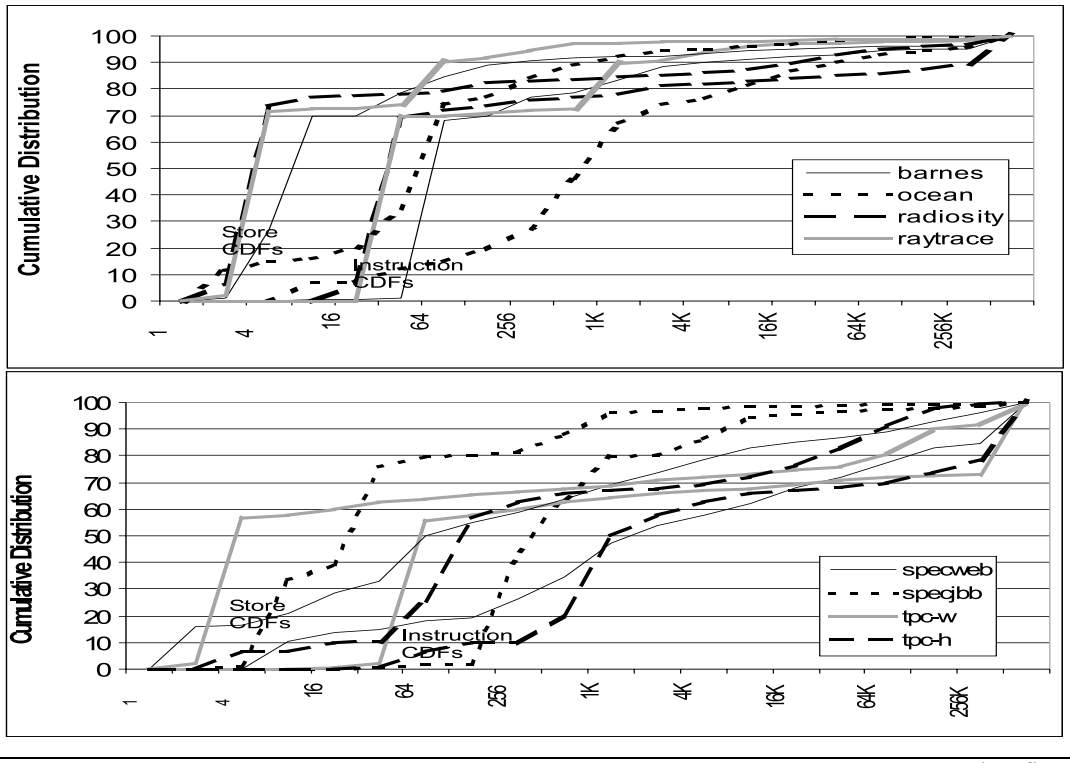

FIGURE 5-10. Dynamic Program Distances Between Useful Temporally Silent Pairs. The figure shows the cumulative distribution of distance (in dynamic instructions and dynamic stores) between the intermediate value store and the temporally silent store in a silent pair for cases in which MESTI prevents a remote miss.

ral silence (Section 5.5.2), we can exploit the dynamic program distance between the intermediate value store and temporally silent store to further reduce the cost of detection. We call this the temporally silent pair *distance*. To first order, this distance determines the difficulty of detecting TSS with finite buffering, since long-distance temporally silent pairs require a deeper memory for tracking the original value.

In Figure 5-10, we show the cumulative distribution of store pair distance measured in terms of both dynamic non-update silent stores and instructions executed for *useful* temporally silent pairs. We call a pair useful if it can be exploited to prevent a communication miss under MESTI. Focusing first on dynamic store distance, in the scientific workloads we see that over 80% of useful silent pairs can be captured within a distance of 64 dynamic stores. However, in the commercial workloads this same distance will

only capture 50% of useful silent pairs (only 25% in *tpc-h*), with the 80% level not passed until distances of 8K, 32K, and 64K in *specweb*, *tpc-h*, and *tpc-w*, respectively. Examining dynamic instruction distance, the trends are similar, but we observe greater separation between benchmarks. In the scientific workloads, a short distance of 32 instructions captures almost 70% of opportunity in *radiosity* and *raytrace*, but the same level is not reached for *barnes* and *ocean* until distances of 128 and 2K, respectively. In the commercial workloads, *tpc-w* reaches 55% of opportunity within distance 64, but for *specjbb, specweb*, and *tpc-h* at least 80% of opportunity is not reached until distances of 1K, 16K, and 32K instructions or more, respectively. In summary, for many cases the temporally silent pair distance can be substantial, especially in commercial workloads. Again, this implies that in-core techniques with limited buffering are unlikely to be effective, thus general mechanisms which can detect temporal silence occurring at these substantial temporally silent pair distances are worthwhile.

However, in most cases, capturing distances on the order of 100K instructions/stores can achieve most of the available opportunity. Therefore, stale storage may not be needed throughout the entire memory hierarchy when this fact is considered; large L1 caches, along with the natural combining of references that such structures provide, may prove effective to capture such distances. We discuss such a scheme, which can detect temporal silence, at essentially no storage cost, in the next section.

5.5.4 Taking Advantage of Inclusive Memory Hierarchies

One efficient method of detecting temporal silence exploits the natural behavior of inclusive cache hierarchies. Consider a writeback L1-D cache with an inclusive L2. When the processor writes a cache line, the line is brought into the L1-D and is written while the

L2 updates its tag array to indicate the line is modified in the L1-D. Note that the L2 data is actually a stale version of the cache line, and it can be used for detecting temporal silence without any explicit stale value storage.[10]

However, a difficulty arises when a comparison against the stale value is required to determine whether temporal silence has occurred—naively, on every store! Obviously such a scenario is undesirable, as the natural write-combining ability the writeback L1-D cache was providing is sacrificed; each dynamic store will require a comparison against the L2 data. The comparison need not be cache line width if sub-block dirty bits are employed, where each bit indicates whether each sub-block has a different value than the L2 data. However, a better solution is desirable.

One solution is allowing write combining at the L1-D cache for some period of time, reducing the number of comparisons required and also elongating the detection time as discussed in Section 5.3. We explore two different possibilities in our writeback hierarchy, one where detection is delayed until dirty lines are written back from the L1-D to the L2, and another where the comparison is performed when a dirty line reaches the not-most-recently-used (NMRU) class in a set associative cache. The NMRU case can be considered similar to the eager writeback proposed for a different purpose by Lee et al. [65] and can have similar benefits on the L1-L2 interface.

In Figure 5-11 we show the reduction in communication misses for the two previously described cases (NMRU and writeback). Three different L1-D cache sizes are shown (8KB, 32KB, and 128KB—all 4-way associative) with infinite L2 caches. Results

10. Special consideration must be made for temporally silent pairs which cross an L1-D writeback event; since the writeback destroys the previous copy of the cache line (with the dirty L1-D data), the L2 data can no longer be used for detecting temporal silence. This is properly accounted for in the simulations presented in this section.

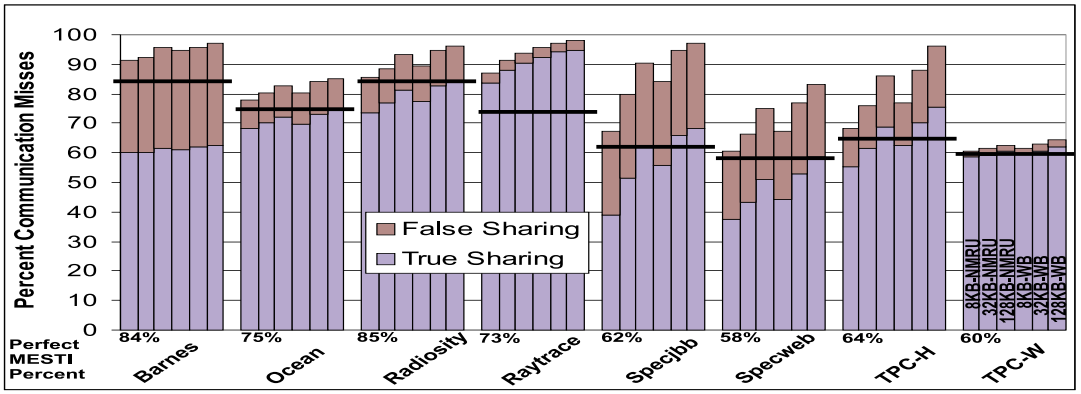

FIGURE 5-11. Communication Misses with Different Inclusive Cache Hierarchies. The fraction of communication misses observed with MESTI for 8KB, 32KB, and 128KB (4-way associative) L1-D caches is shown. Temporal silence is detected when the cache line reaches the NMRU class and when the line is written back (WB) to the L2 cache. The percentage of communication misses with perfect MESTI is indicated numerically and with the solid lines for each benchmark; schemes approaching the solid line are better.

for perfect MESTI are indicated with the dark lines across each benchmark. These results indicate the timeliness of each approach, as discussed in Section 5.3. Focusing first on the NMRU results, we see that this policy foregoes little opportunity for a small L1-D cache; in the 8KB case, *raytrace* is the only exception. As cache size increases, more opportunity is sacrificed (comparing the first three bars from the left for each benchmark). This is intuitive; as cache size increases, fewer temporally silent stores are detected in a timely fashion because the delay for a cache line to become NMRU increases. Comparing the NMRU results to the writeback (WB) results shows a similar, although more pronounced, trend.

Behavior for each benchmark varies greatly; *tpc-w* shows essentially no sensitivity to cache size or policy (NMRU vs. WB), while the other commercial workloads can sacrifice significant opportunity for a 32KB, NMRU policy, e.g. *specjbb*. This result can be interpreted in different ways. In the case of small L1-D caches, it may be possible to exploit temporal silence using the MESTI protocol by only making changes to the L1-L2 interface and the L2 cache of existing designs (based on the WB results), thus avoiding core redesign and achieving tangible benefit. However, for medium or large L1-D caches,

slight modification to the L1-D cache is most likely necessary to avoid losing most of the opportunity. Note also that many factors contribute to the behavior observed for each benchmark: the temporally silent pair distance (which cannot cross an L1-D writeback event), L1-D cache working set size for the benchmark, and also the benchmark's sensitivity to timeliness of detecting temporal silence. We characterize each benchmark's sensitivity to timeliness in Section 5.10.

Finally, note that the WB results roughly correspond to the NMRU results for their respective 4x larger cache; this is also intuitive as the MRU class of a 32KB, 4-way associative cache is approximately 8KB. Thus, despite slightly different index functions, the NMRU class of the 32KB cache is equivalent to writebacks from an 8KB cache.

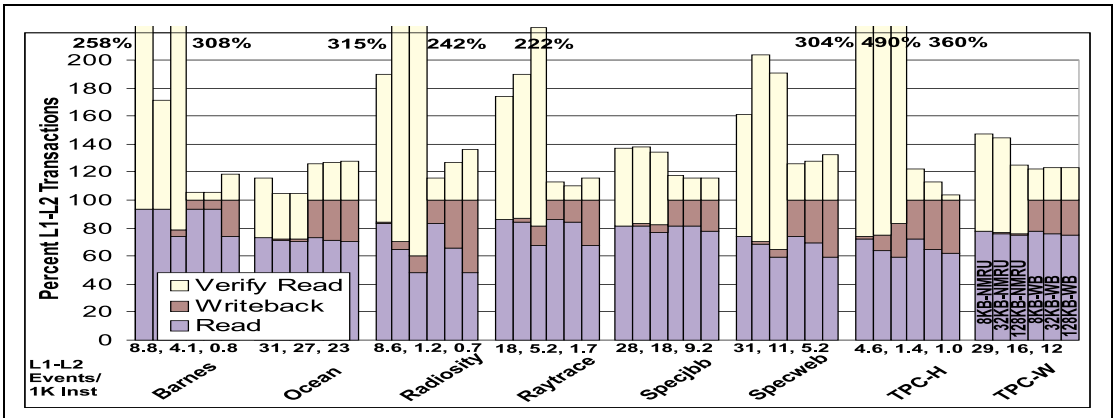

FIGURE 5-12. L1-D Cache to L2 Transactions for Different Cache Configurations. The fraction of L1-D cache to L2 transactions due to Read misses, Writebacks, and additional read requests for detecting temporal silence (Verify Read) are indicated for 8KB, 32KB, and 128KB (4-way associative) caches for NMRU and WB. The Writeback category includes dirty replacements and modified interventions. Graphs are normalized to the baseline case for that cache configuration, with the L1-L2 event rate shown numerically.

Exploiting inclusion between the L1 and L2 caches to obtain zero-space detection of temporal silence is not without a cost, namely additional transactions across the L1-L2 interface. In Figure 5-12, we show the breakdown of L1-L2 transactions for each scenario described previously. Results for each configuration are normalized to the baseline trans-

action rate for the particular cache hierarchy to make the scale of the graph readable. Note that the Writeback category includes both dirty castouts and modified interventions. Therefore, Verify Read can be less than Writeback since modified interventions are not checked for temporal silence. Furthermore, the NMRU results for Writeback are solely due to modified interventions, NMRU Verify Reads are considered eager writebacks, thus cleaning the cache line [65].

Examining writeback temporal silence detection, we observe that in most cases, the contribution of additional transactions for verification is less than 30%. This additional pressure on the L1-L2 interface might be accommodated in existing designs without significant difficulty. The number of writeback buffers may need to be increased as the occupancy for each is longer; the data must be held in the buffer for both the verify-read and the write back operation if temporal silence is not detected.

Examining NMRU temporal silence detection, we observe that in many cases, additional traffic across the L1-L2 interface is substantial—up to an additional 490% in *tpc-h* for a 32KB cache. However, this benchmark is most likely not the worst case since the baseline L1-L2 transaction rate is low compared to other commercial benchmarks. Examining *specweb*, which has relatively high baseline L1-L2 bandwidth requirements for 8KB and 32KB caches, we observe increased bandwidth demands of 60% and 100%, respectively. Although these increases are much smaller than detecting temporal silence on every non-update silent store, i.e. a write-through hierarchy with no write combining, the extra bandwidth demand is substantial. Therefore, to obtain both timely detection (the NMRU results from Figure 5-11) and zero-space overhead by exploiting inclusive hierarchies, we need a better method for filtering which requests should be checked for temporal

silence. Various prediction mechanisms might be useful here; instead, we propose a very simple, non-speculative, and very effective method for filtering verification. We will discuss it in Section 5.5.6.

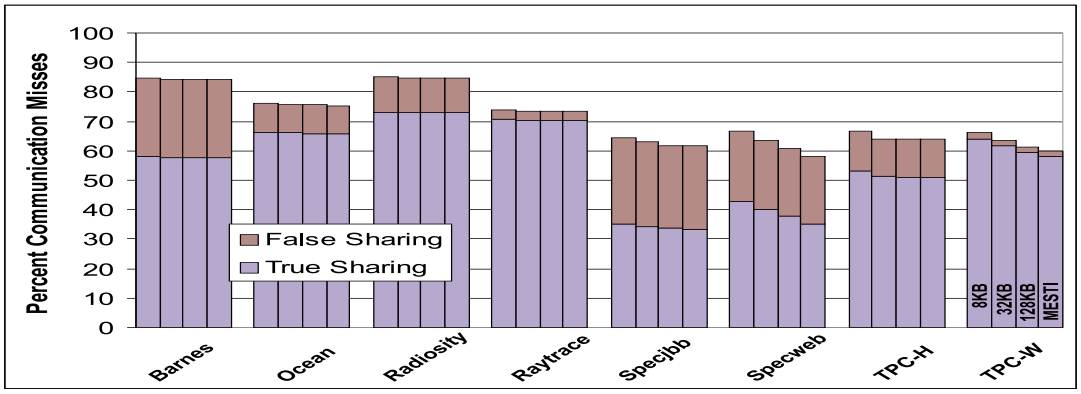

FIGURE 5-13. Communication Miss Opportunity Lost Due to L1-D Cache Writebacks. The fraction of communication misses observed with MESTI for 8KB, 32KB, and 128KB (4-way associative) L1-D caches is shown. Temporal silence is detected immediately, but only if an L1-D cache writeback has not occurred between the temporally silent pair. The percentage of communication misses with perfect MESTI is also indicated; schemes approaching the MESTI bar are better.

We have noted at various points throughout this section that temporally silent pairs which cross L1-D writeback events cannot be captured with this zero-space overhead detection mechanism; dirty L1-D castouts overwrite the previous value in the L2 cache, and thus we can no longer use the L2 cache data for detecting temporal silence. All data presented previously considers this, but also delays detection. To indicate communication miss opportunity lost solely due to this effect, we show observed communication misses for different cache hierarchies for immediate temporal silence detection in Figure 5-13. We can see from the figure that capturing temporal silence across L1-D cache writebacks for the configurations studied (8KB, 32KB, 128KB, 4-way associative) is unnecessary for the scientific workloads. For the commercial workloads, most opportunity is still captured even with a small 8KB L1-D cache; however, *specweb* and *tpc-w* show non-trivial opportunity lost (6% and 8% of communication misses, respectively). This result agrees with

Figure 5-10, as both *specweb* and *tpc-w* show a non-trivial fraction of long-distance temporally silent pairs. However, *tpc-h* does not show the same behavior, indicating that although the silent pair distance can be significant in this benchmark, the working set of lines touched within the store pair is small. This conclusion is further strengthened by the results of Figure 5-1 which shows *tpc-h* having the fewest capacity misses for the finite cache configurations measured.

5.5.5 Adding Explicit Stale Storage Designed To Detect Temporal Silence

Utilizing inclusive hierarchies as the only storage for detecting temporal silence may not be fruitful, or even possible, under some circumstances. For example, we saw in the previous section that commercial workloads which exhibit large data working sets may have temporally silent store pairs longer than the L1-D lifetime for small caches, and restricting detection to be a function of L1-D cache configuration may lead to performance anomalies on other workloads. Furthermore, this technique will not work for write-through hierarchies because each store must be reflected in the L2 when it is performed.[11] Therefore, we would like to devise a more general method of adding explicit storage for detecting temporal silence across L1-D cache writebacks. We discuss the proposed technique in the context of a writeback hierarchy, but it is extensible in a straight forward way for write-through hierarchies. We assume that global coherence state is maintained at the L2 cache.

A block diagram of the proposed mechanism is shown in Figure 5-14. The basic principle of operation is simple: whenever the L1-D cache displaces a dirty line, the previ-

11. Combining write buffers and write caches can reduce the actual number of store-throughs required in these types of hierarchies, but a similar argument applies in these cases as well. A discussion of trade-offs between hierarchy types considering error detection and correction techniques has already been presented in Section 2.3.3.

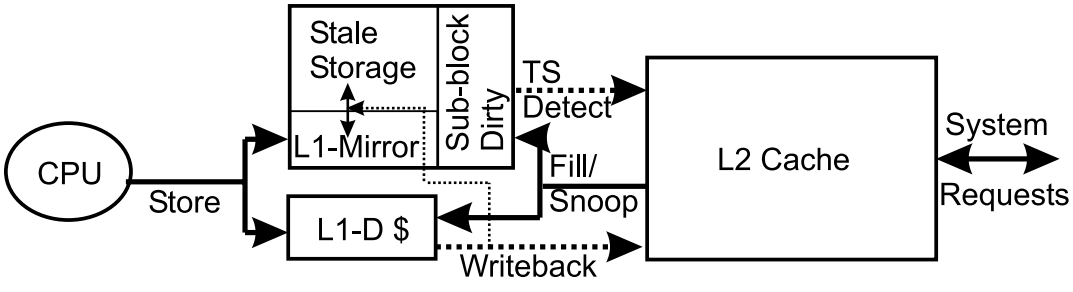

FIGURE 5-14. Illustration of an Efficient Stale Storage Mechanism. Capturing temporally silent pairs across L1-D cache writebacks requires additional storage, which can be managed as shown.

ous version of the line is saved as the candidate for future temporal silence detection. This is done in the stale storage shown in the figure. However, obtaining the correct stale value on the first writeback requires reading data from the L2; the L1-D writeback data cannot simply be captured since it is already dirty, the intermediate value is already stored within the cache line. In order to rectify this, the writeback can simply be converted into a read-write operation, doubling the required L1-L2 interface bandwidth. Another option, shown in the figure, is to add an explicit L1-Mirror which captures the stale value when the cache line is initially filled into the L1-D cache by either a load or store miss. If the cache line is subsequently modified and written back, the data from the L1-Mirror is copied into the stale storage and the dirty data is written in the L2, avoiding the L2 read-write sequence. Note that the stale data storage is only required for dirty L1-D castouts which have not exhibited temporal silence within their lifetime, thus its capacity for capturing temporal silence is greater than simply enlarging the L1-D cache and exploiting inclusive hierarchies as described in Section 5.5.4.

A slight complication arises in design of the L1-Mirror and its interaction with the stale storage and L2 cache. Namely, the L1-Mirror should know when to capture incoming data from the L2 on a fill or use stale data as the candidate for temporal silence. Sometimes the incoming data from the L2 is the correct stale value, corresponding to the labeled

arcs in Figure 5-6 on page 118 which indicate when candidates for reversion should be saved; other times it is not because of a previous writeback of an intermediate value. Fortunately, this can be rectified at the L2 level; when the fill occurs, the L2 simply informs the L1 whether the line had been previously written back, i.e. it is in M state at the L2, or whether the fill corresponds to a correct stale version (the labeled arcs in Figure 5-6). With this knowledge, the L1-Mirror either captures the L2 fill data or copies the data from the stale storage and uses it for detection. The datapaths between the L1-D cache and L2 cache behave as normal—the most up-to-date copy of the data either resides in the L1-D cache or in the L2 cache at all times, thus external snoops need not access the stale storage or L1-Mirror to service interventions.[12]

Note also that this structure allows detection of temporal silence without an L1-L2 read transaction for each store as the simple method of Section 5.5.4 employed. Each store can simultaneously write into the L1-D cache and also compare its value against the L1-Mirror to detect temporal silence with perfect timeliness, achieving the maximum reduction in communication. Temporal silence for an entire cache line is indicated by the nor of all sub-block dirty bits, as shown in Figure 5-14.[13] Note that update silence may not be detected in general by exploiting the L1-Mirror, as stores which are silent with respect to the L1-Mirror may not be silent with respect to the L1 because of previous L1 writebacks. However, if we are only interested in exploiting update silent stores for communication miss reduction (and not for core performance enhancement as discussed in Chapter 2), the L1-Mirror may be used for this purpose; we describe such an approach later in Section

12. The stale storage or L1-Mirror may or may not need to be snooped on remote invalidates or L2 castouts depending on the detailed protocol design.
13. The sub-block dirty bits must be kept at the smallest write atom (a byte in most architectures) for this technique to be valid.

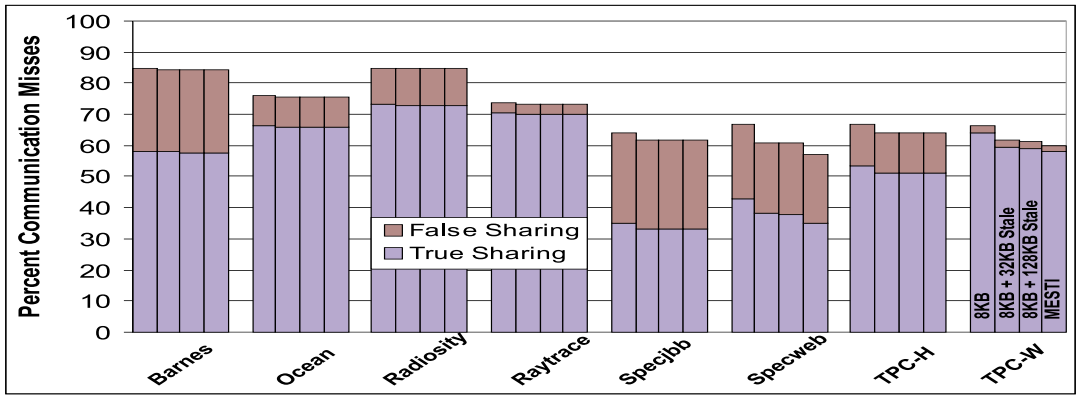

FIGURE 5-15. Communication Misses for Different Combinations of Stale Storage. Communication misses for an 8KB (4-way associative) cache exploiting only inclusive hierarchies for temporal silence detection is compared to the same 8KB cache augmented with either 32KB or 128KB of stale storage. Results for MESTI with full stale storage are also indicated. The cost of the L1-Mirror is an additional 8KB in all cases.

Figure 5-15 shows the ability of this structure to capture useful temporally silent pairs under MESTI for a small 8KB, 4way, L1-D cache for stale storage capacities of 32KB and 128KB. We see that both stale storage capacities are effective at capturing useful temporally silent pairs across all benchmarks. This is not surprising, as an inclusive hierarchy with 128KB L1-D cache (Section 5.5.4) was also able to capture the vast majority of the potential. However, explicit stale storage of 32KB in combination with an 8KB L1-D cache performs better than a 128KB L1-D cache for all benchmarks. As explained previously, this is possible because only dirty L1-D castouts need be tracked in the stale storage. Furthermore, this technique creates no additional delay in detection (Section 5.3) via the L1-Mirror, thus perfect timeliness is achieved. However, this timeliness comes at the cost of each store comparing its value against the L1-Mirror immediately. Write combining techniques, such as those described in Section 5.5.4, can reduce the number of comparisons with some timeliness cost. However, note that comparisons are only performed against the L1-Mirror, and not the Stale Storage; thus the L1-Mirror can be con-

structed to have similar access time to the L1-D cache since the configurations are identical. The Stale Storage can be slower as it is only written on L1-D cache writebacks, read on L1-D cache fills, and data can always be replaced from it with no correctness issue because correct coherent copies of the cache line are always either in the L1-D cache or the L2.

This method of temporal silence detection has extremely low space cost (the size of the L1-Mirror plus Stale Storage). We stated in Section 5.5.2 that naively limiting storage throughout the entire memory hierarchy, reserving 1.8MB of stale storage with a 14MB L2 was more effective at reducing misses than a 16MB cache without temporal silence exploitation. The configuration shown in this section (8KB L1-Mirror and 32KB Stale Storage) has added only 40KB/16MB = 0.25%, and can achieve approximately a 40% reduction in communication misses for commercial workloads. Obviously, adding such storage is more beneficial than adding 40KB of additional L2 cache space. For large L1-D caches, much space may be replicated unnecessarily in the L1-Mirror; in these cases simply utilizing inclusive hierarchies (Section 5.5.4), or foregoing the L1-Mirror may be more desirable.

5.5.6 Eliminating Unnecessary Comparisons with Existing Value Summaries

In Section 5.5.4 and Section 5.5.5 we discussed utilizing inclusive hierarchies and explicit storage for low space-cost temporal silence detection. However, these mechanisms required either a large increase in L1-L2 transactions (Section 5.5.4) or many explicit comparisons against dedicated storage (Section 5.5.5) to enable low-space overhead.

Many architectures now include explicit ECC protection on all levels of memory

hierarchy due to the increasing presence of soft errors, as discussed at length in Section 2.3.3. ECC provides a mechanism through which single bit errors in the ECC-word can be corrected and double-bit errors detected if implemented with SEC-DED codes [11, 97]. A key observation is that ECC provides a *value summary* of data stored within the ECC-word—an efficient hash of the value. This value hash can further reduce the cost of detecting temporal silence.

For both of the architectures described in Section 5.5.4 and Section 5.5.5, the number of comparisons can be reduced by simply gating a full-comparison against the stale value using the ECC bits. Since a given data value is always represented with the same ECC bits for common mapping functions, for data values to match, the ECC hash must be identical. Therefore, a full data value comparison for detection is only required when the ECC hash of the current value matches the stale value. Note that it is also possible to use a subset of the L2 ECC-bits for comparison against the L1 ECC-bits to limit additional storage overhead.

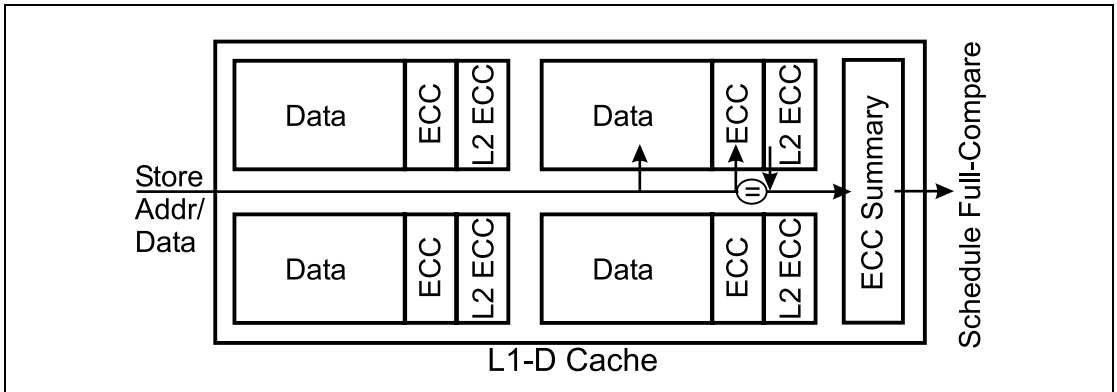

FIGURE 5-16. L1-D Cache Modified to Exploit ECC Information. The figure shows data paths exercised on each store operation. A full-comparison is only scheduled if the ECC Summary indicates that ECC hash values match for all sub-banks between the L1-D cache and the L2.

A diagram for inclusive, write-back hierarchies is shown in Figure 5-16. A similar architecture is possible for explicit stale storage. We assume the cache sub-bank size

matches ECC-word size for ease of illustration. On each committed dynamic store operation, the value is written into the data array and the ECC bits are updated. Simultaneously, the L2 ECC value is read and compared against the new ECC hash; the status of this comparison is stored in the ECC Summary, a single bit for each sub-bank/ECC word. If all ECC-words match for the entire cache line, as recorded in the ECC Summary, an full-comparison of the L1-D cache data and the L2 data (stale value) is scheduled, and is performed as described in Section 5.5.4. The other write combining techniques described in that section can be used in concert with the mechanism described.

The architecture described may not affect critical cache timing paths for the following reasons: no modification beyond a standard ECC-protected L1-D cache is required for the data and ECC arrays. The L2 ECC values are only read on each dynamic store, and are written only on L1-D cache fills (with the stale version ECC value), and can be tracked on sub-bank granularity. Finally, even though the ECC Summary must be updated from any sub-bank on every cycle, updates to it can be pipelined and delayed as it is not used for correctness, but only performance optimization.[14] Finally, we assume the same ECC mapping functions are used at both the L1 and L2 level; it may be possible to use different mapping functions or ECC-word sizes for each level provided the chosen mapping functions provide a subset of ECC bits which can be compared between levels.

We show the ability of this approach to reduce explicit comparisons between the L1-D cache and L2 for inclusive hierarchies in Figure 5-17 for both 8-bit ECC hashes (64-bit ECC data words) and a 4-bit subset of the same ECC hash. Results for comparisons

14. In microarchitectures that achieve multi-ported L1-D caches through multi-banking as opposed to explicit multi-porting, the ECC Summary must be multi-ported as well and cannot be banked for most common bank mappings. However, for architectures that can commit multiple stores per cycle, the cache tags must already be multiported for low-order bank interleaving, so complexity for these two structures will be similar.

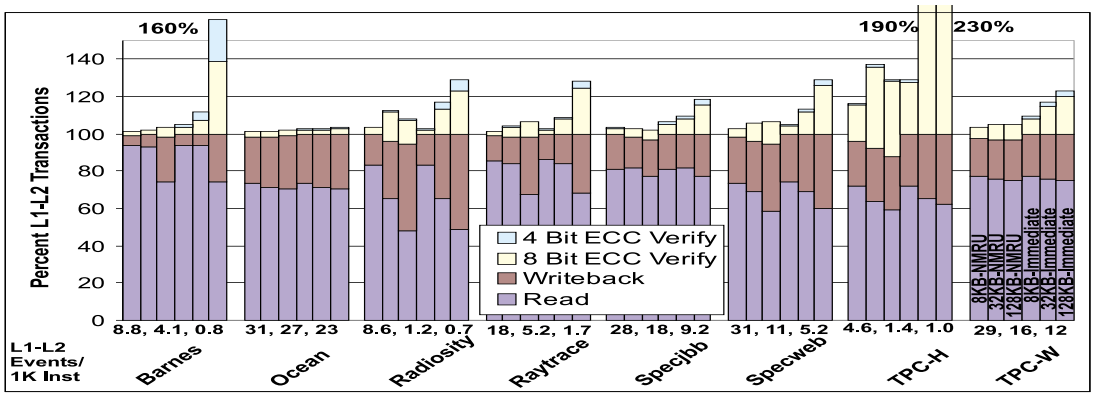

FIGURE 5-17. L1-D Cache to L2 Transactions Exploiting ECC Value Summaries. The fraction of L1-D cache to L2 transactions due to Read misses, Writebacks, and additional read requests for detecting temporal silence using either 8-bit or 4-bit ECC filtering functions are indicated for 8KB, 32KB, and 128KB (4-way associative) caches. Results for detecting temporal silence when the cache line reaches the NMRU class and for immediate detection are shown. Graphs are normalized to the baseline case for that cache configuration, with the L1-L2 event rate shown numerically.

performed when a cache line reaches the NMRU class within the cache (Figure 5-12) and immediate comparisons are indicated for 8KB, 32KB, and 128KB L1-D caches. Comparing the NMRU results from Figure 5-17 to Figure 5-12 where no value summary information is used, we see a dramatic reduction in utilized L1-L2 bandwidth across all benchmarks. All cases (except *tpc-h*) show less than 15% additional bandwidth for temporal silence detection. Turning to the immediate detection results, for small L1-D caches (8KB and 32KB), all increases in L1-L2 traffic are less than 20% (except for *tpc-h*). Note that the increase is linear in base L1-L2 transaction rate for immediate detection as the number of dynamic stores which cause ECC hash function matches is identical across cache configurations; therefore, the larger contribution to traffic comes from the lower miss rate of the baseline cache hierarchy. Finally, we note that additional comparisons when using only 4 bits of the ECC hash are minimal (except *barnes* 128KB), indicating that using a subset of the L2 ECC for comparison may be worthwhile to limit storage overhead. The storage overhead for each scheme against a conventional ECC-protected cache can be computed simply as:

(L2-ECC-bits + 1 (ECC Summary)) / (ECC-word-size + ECC-bits)

For the caches indicated, the storage overhead is (8 + 1) / (64 + 8) = 12% and (4 + 1) / (64 + 8) = 7%, respectively.

Overall, we see that using ECC value summaries significantly reduces the bandwidth required for detection over the L1-L2 interface in most cases. In one case, *tpc-h*, the additional bandwidth can still be substantial; however, the base transaction rate is low, indicating that the extra bandwidth demand may be acceptable. Furthermore, using ECC summaries can practically enable instant detection in a write-back hierarchy by limiting extra traffic on the L1-L2 interface. Results for sharing miss reduction with both methods have been presented previously and are equivalent to the NMRU and MESTI results in Figure 5-11.

5.6 Critical Temporal Silence

We have shown throughout previous sections that temporal silence can be exploited to effectively eliminate communication misses. However, exploiting temporal silence with explicit communication can have an effect on address transactions observed in the system. We discussed similar concepts in Section 4.4 on page 84 in the context of critical update silent stores in multiprocessors. Similar concepts apply with temporal silence; therefore we use the same terms defined in Section 4.4. As a brief recap, we term a specific dynamic temporally silent store critical if exploiting it leads to a reduction in communication misses, anti-critical if modifying its behavior with respect to the baseline case leads to a net address transaction increase, and non-critical if modifying its behavior has no impact.

As a case study, let's consider our MESTI protocol (Section 5.4.2), which relies on

explicit communication through the added T state and validate transactions. Each validate has the potential to eliminate multiple remote cache misses; thus MESTI has the potential to reduce overall address traffic. However, each validate also requires the validating processor to forego exclusive access to the cache line; thus if a subsequent non-update silent store occurs to the line, an upgrade is required. If a remote miss was not eliminated by the validate, less overall address traffic would have resulted by simply maintaining exclusive ownership instead of broadcasting the validate. To make this scenario more concrete, we present Table 5-4.

Table 5-4: Illustration of Critical Temporal Silence in Multiprocessors. The Validate/Upgrade pair at T2 and T3 creates additional address transactions as compared to the baseline case because the Validate relinquishes exclusive ownership of the cache line, requiring an Upgrade to obtain write permission. The baseline execution is shown in the bottom half of the example.

	CPU 0		CPU 1	
Time	Instruction	Cmd/Txn	Instruction	Cmd/Txn
T0	LD [A] (0)	Read	LD [A] (0)	Read
T1	ST [A], 1	Upgrade		
T2	ST [A], 0	Validate		
T3	ST [A], 2	Upgrade		
T4			LD [A] (2)	Read
Baseline Execution Without MESTI Protocol Enhancement				
T0	LD [A] (0)	Read	LD [A] (0)	Read
T1	ST [A], 1	Upgrade		
T2	ST [A], 0	<None>		
T3	ST [A], 2	<None>		
T4			LD [A] (2)	Read

The execution observed with MESTI is shown in the top half of the table; for comparison purposes, the baseline execution is shown in the bottom. Notice the additional validate and upgrade transactions at T2 and T3. These occur because the temporally silent store at T2 is not the last write [64], i.e. the cache line is re-written before a remote access occurs. In the terminology used throughout Chapter 5, we call the validate at T2 *useless* because it does not prevent a remote miss. Furthermore it is anti-critical because this write

is not a last write; changing system behavior to exploit it is harmful because it creates additional address transactions. As discussed in detail in Section 4.4 for update silent stores, anti-critical temporally silent stores can have a two-fold impact on system performance: additional address traffic is created increasing queuing delays for other coherence transactions and the validate/upgrade pair can cause commit of the store at T3 to be delayed waiting for write permission.

As in Section 4.4, we can define a temporally silent store as critical, anti-critical, or non-critical. In discussions that follow, we group anti-critical and non-critical under the term *useless* as both classes indicate transactions which do not lead to sharing miss reduction. Anti-critical temporally silent stores can have both negative performance impacts described, non-critical will only have the former, by definition. Note that anti-critical temporally silent stores always lead to the validate/upgrade pair illustrated in Table 5-4. We will explore the prevalence of anti-critical temporally silent stores by measuring the percentage of temporally silent stores which are not last writes, as well as discuss the impact these anti-critical temporally silent stores have on address traffic in Section 5.7.

In order to gain further insight into the program behavior and also the predictability of critical temporal silence, we can explore the dynamic memory footprint of stores exhibiting critical temporally silent behavior. Understanding this working-set size enables design of space-efficient prediction mechanisms. The working set for both scientific and commercial workloads is shown in Figure 5-18.

In the scientific workloads, we observe that approximately a 2KB working set of memory locations contributes over 80% of all TSS avoidable misses in *barnes*, *ocean*, and *raytrace*; with *radiosity* a 20KB working set is required. The commercial workloads

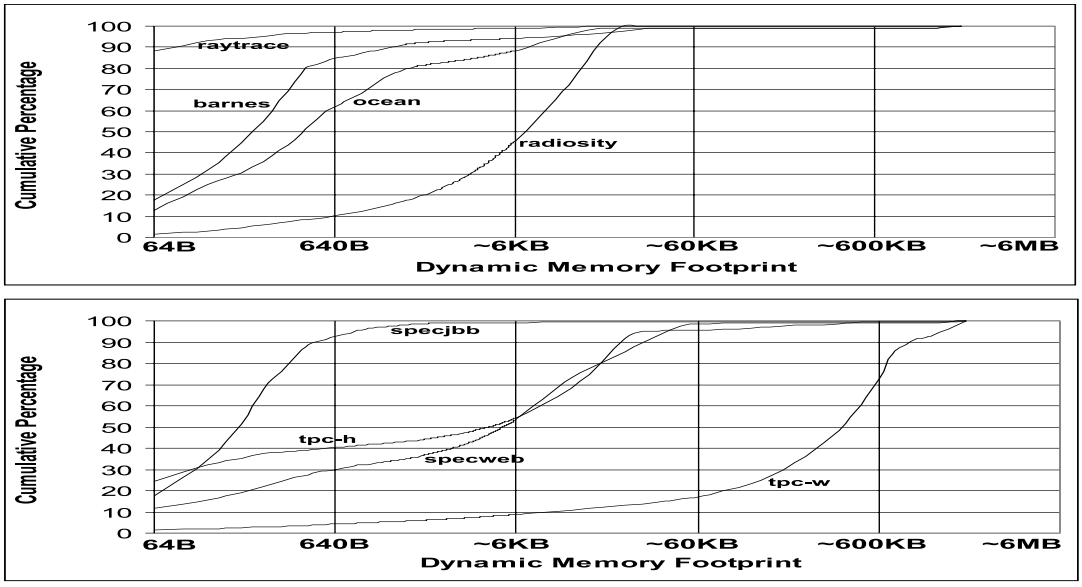

FIGURE 5-18. Dynamic Memory Footprint Contributing to Critical Temporal Silence. The cumulative distribution of memory addresses contributing TSS-avoidable misses is indicated. The scientific workloads are shown in the top graph, commercial workloads in the bottom graph. Notice the log scale on the x-axis for both graphs.

exhibit a larger working set in general (nearly 1MB is required in *tpc-w* to reach 80% of opportunity captured, for example) and there is more variation between workloads. This is reasonable given the observation by many researchers [96, 75, 9] of increased memory footprints in commercial workloads.

In contrast to USS (Section 4.4) we also observe that the dynamic footprint of memory locations exhibiting critical temporal silence is not strictly proportional to the fraction of sharing misses which are eliminated under TSS; in comparing Figure 5-5 on page 117 with Figure 5-18, we observe that no strong correlation exists between working set size and TSS avoidable communication misses. This indicates that benchmarks reaping greater benefit under TSS may do so either by sharing a large number of cache lines temporally silently, e.g. *tpc-w*, or a small number frequently, e.g. *specjbb*. Therefore, mechanisms attempting to capture temporally silent behavior must consider both large working sets and high-frequency sharing to be most effective.

As described in the previous sections, exploiting temporal silence with explicit communication using the MESTI protocol has many advantages in implementation and can capture most TSS avoidable misses. However, as discussed in Section 5.6, naive MESTI implementation may have an adverse effect on address transactions. We explore this effect in this section by studying efficient communication mechanisms.

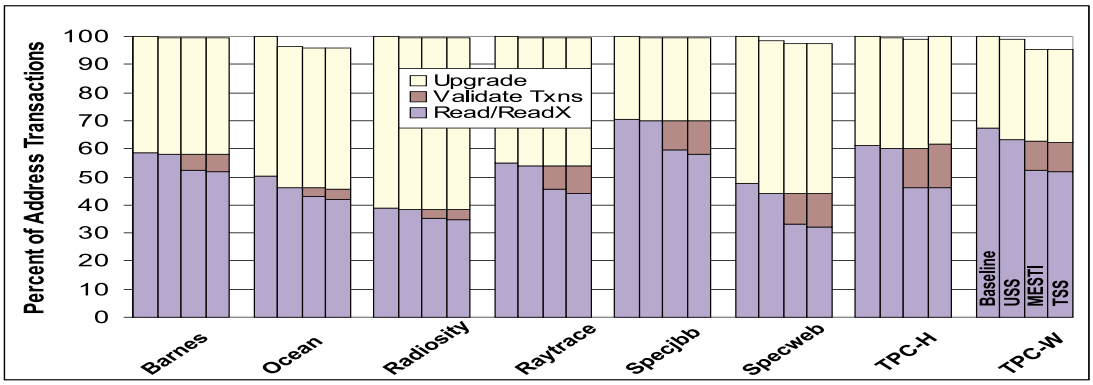

FIGURE 5-19. Best Case Address Traffic Exploiting Temporal Silence. Address traffic increases for USS, MESTI, and TSS are shown for an oracle predictor of critical update silence and critical temporal silence.

In Figure 5-19, we present the best possible performance of MESTI and TSS with respect to observed address traffic in the system. The stacked bars indicate the percentage of address traffic in the system, normalized to the baseline, due to data requests (Read/ReadX), additional requests due to temporal silence exploitation (Validate), and finally ownership (Upgrade) requests. In the figure, we assume an oracle predictor for useful validates; a validate is only broadcast if at least one remote processor is able to avoid a data transaction due to it. For all benchmarks, overall address traffic remains essentially constant or decreases slightly. This result is reasonable, as a single validate can avoid multiple demand data transactions. Note further that upgrade requests remain essentially constant across configurations, with validates simply replacing Read/ReadX transactions.

.

Since a validate can only re-install a cache line in remote caches in shared state, this indicates that validates are eliminating remote read misses as opposed to write misses; otherwise removed Read/ReadX transactions would have a corresponding increase in upgrades for MESTI and TSS.[15]

Table 5-5: Address Traffic Increase and Last Write Statistics. Results are shown for MESTI with infinite, 16MB, 8MB, and 4MB (8-way associative) caches. Last Write Accuracy is measured for an infinite cache.

Benchmark	Naive Validate				Snoop-Aware Validate				Last Write Accuracy
	Inf.	16MB	8MB	4MB	Inf.	16MB	8MB	4MB	
barnes	15.3%	15.3%	15.3%	15.3%	15.3%	12.6%	12.6%	12.6%	5.80%
ocean	45.3%	31.5%	30.0%	17.6%	38.7%	24.0%	22.2%	9.01%	15.4%
radiosity	52.4%	35.1%	32.3%	31.0%	43.8%	27.7%	25.5%	24.3%	14.3%
raytrace	70.2%	32.8%	32.8%	31.5%	57.9%	20.0%	20.0%	19.2%	24.9%
specjbb	107.7%	53.9%	49.2%	44.9%	97.3%	40.6%	36.5%	32.0%	20.0%
specweb	92.2%	52.7%	47.7%	41.2%	88.8%	45.4%	39.2%	32.0%	24.1%
tpc-h	239%	223%	218%	214%	225%	219%	214%	209%	11.3%
tpc-w	80.6%	55.8%	39.9%	25.6%	77.6%	52.2%	34.7%	17.9%	26.0%

Examining a more realistic scenario, in Table 5-5 we show the measured increase in address traffic over Baseline when broadcasting a validate at each temporal silence detection (Naive Validate) for differing cache configurations. In most cases the address traffic increases considerably over the ideal case in Figure 5-19, motivating attempts at reduction. A simple way to reduce the traffic is to collect snoop responses to ReadX/Upgrade transactions indicating whether or not the cache line was present in a remote cache at the time. If the responses indicate it was not present in any remote cache, it is certain that any validate for this line is useless. We call this policy *Snoop-Aware Validate*, with its performance indicated in the table[16]. The reduction in address traffic due to this simple optimization is non-trivial in *ocean*, *radiosity*, *raytrace*, and *specjbb* in the case

15. Address transactions for TSS reflect a hypothetical result, since we have not described a mechanism to exploit all TSS misses through address transaction broadcasts.
16. Figure 5-6 does not show support for this optimization to reduce clutter.

of infinite caches.

In the case of finite caches, we see that the address traffic increases for Naive Validate are significantly less than the infinite case, primarily due to two factors: 1) Absolute increase in baseline address traffic due to added capacity/conflict misses; 2) Fewer Validates because lines which have been replaced from the cache between the intermediate value store and the temporally silent store do not lead to a Validate, or prevent a remote miss, because stale storage is not added in memory. More noteworthy is the relative effectiveness of the Snoop-Aware Validate policy in the case of finite caches. All benchmarks except *tpc-h* show a non-trivial reduction in additional address transactions due to Snoop-Aware Validate; in all workloads except *tpc-h* (all cases), *barnes,* and *tpc-w* (for the 16MB case) the reduction is at least 7%.

The substantial increase in address traffic can be understood by examining how often a temporally silent write is the last write to the cache line of interest, where the last write is the final write to the cache line before it is requested by another processor [53]. We have discussed this concept and its impact on address transactions in exploiting temporal silence in Section 5.6. From the table, we see that the last write accuracy of temporal silence is always less than 26%. Therefore, many anti-critical temporally silent stores occur in these workloads leading to what we call *address thrashing* (many unnecessary validate/upgrade pairs). The negative performance impact of address thrashing may be substantial if not actively prevented, largely offsetting the reduction in communication misses via MESTI.

5.7.1 Reducing Address Traffic by Delaying Validates

One way of reducing address thrashing is to place any outbound validate into a

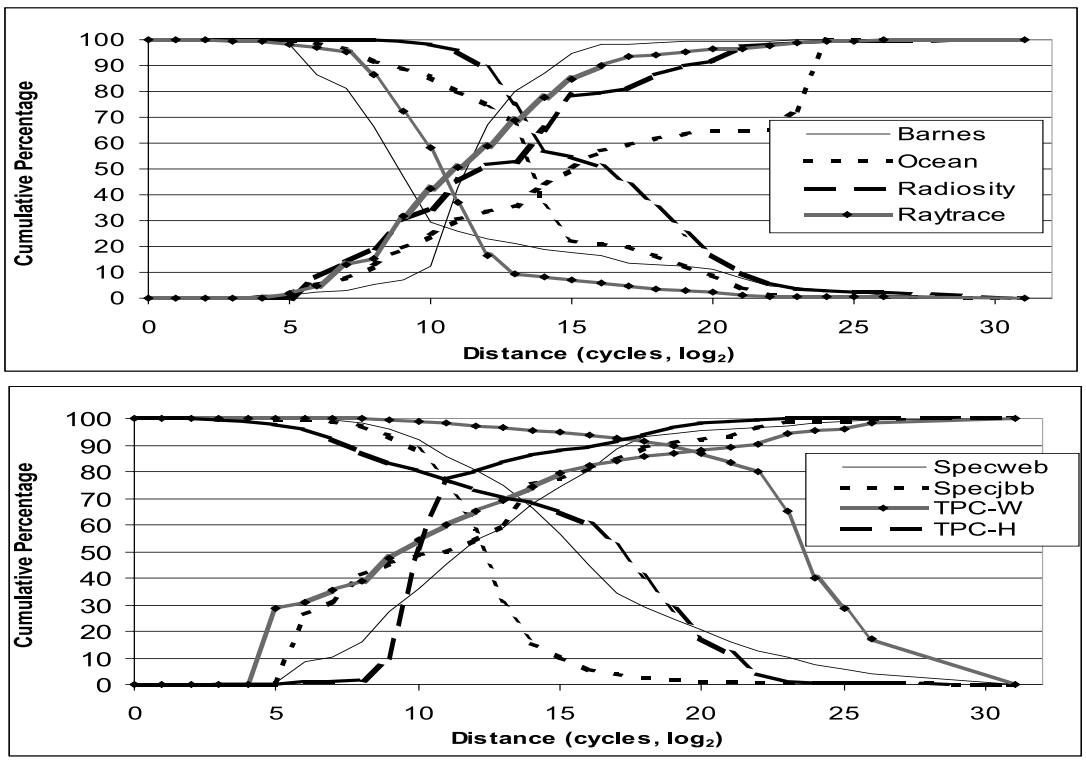

FIGURE 5-20. Temporally Silent Write to Fetch/Next Write Histograms. The monotonically increasing curves indicate the cumulative distance (in cycles) from a temporally silent write to a subsequent non-update silent write for useless cases of temporal silence. The monotonically decreasing curves indicate the cumulative distance (in cycles) from a temporally silent store to a subsequent MESTI avoidable miss.

queue with fixed delay. The validate remains in the queue until the delay time expires and the validate is broadcast, until a demand transaction occurs for the line and temporal silence is communicated to the requestor, or until a non-update silent write occurs to the cache line and the validate is dropped. Delaying validates in this manner filters validates when a non-update silent store is detected to the cache line in the queue, but timeliness of the validates will also be affected, as discussed in Section 5.3. Figure 5-20 characterizes the effectiveness of this delay queue approach. For each benchmark, the monotonically *increasing* curves indicate the distance, in processor cycles, from the final temporally silent store to the cache line to a subsequent (non-update silent) write for cases in which the temporally silent write is not the last write. This cumulative distribution indicates how

many useless validates can be avoided by delaying them by a fixed number of cycles. The monotonically *decreasing* curves indicate the cumulative distribution of the distance, in processor cycles, from the final temporally silent store to a cache line to a subsequent MESTI avoidable miss to the cache line. This cumulative distribution indicates how long a validate can be delayed and still avoid a communication miss.

Ideally, the overall distribution would be bi-modal, i.e. the not last write distance would be short and the MESTI avoidable miss distance would be long. In this case, we could simply build a delay queue long enough to capture most of the not last write distance, and trade relatively few useful validates for this while still allowing useful validates plenty of time to propagate to remote processors. For the scientific workloads, the figure indicates a short queue (2^7 cycles) can eliminate approximately 15% of address thrashing, with approximately 5% of temporal silence opportunity lost in *ocean*, *radiosity*, and *ray-trace*. However, the distribution is not sufficiently bimodal for this approach to eliminate the majority of address thrashing without sacrificing most of the opportunity. For the commercial workloads a short queue (2^7 cycles) removes 30%-35% of address thrashing in *specjbb* and *tpc-w* with less than 1% of opportunity lost. The distribution is sufficiently bimodal in *specweb* and *tpc-w* (at 2^{13} cycles) and *tpc-h* (at 2^{10} cycles) to allow at least 60% of thrashing to be removed with lost opportunities of only 25%, 5%, and 20%, respectively.

Finally, we note that the MESTI avoidable miss distribution indicates that neglecting address bus contention in the characterization results presented throughout Chapter 5 does not affect timeliness of useful validates significantly. The vast majority of useful validates, if sent immediately, will have sufficient time to reach a remote processor. We dis-

cuss correctness issues in delay queue implementation in Section 5.8.3 and Section A.1.

5.7.2 Predictive Snoop-Aware Validate

We can also turn to predictive methods to avoid useless validates and address thrashing. Many works have explored coherence prediction and have determined that many events, such as repetitive block request patterns or migratory data, can be reliably predicted [64] [59] [60] [86] [53] [74]. We explore a simple prediction mechanism, which relies on an augmented form of the snoop-aware validate policy discussed in Section 5.7. Recall that snoop-aware validate simply collects snoop responses at each ReadX/Upgrade to determine whether any remote node had a valid copy of the cache line at the time of the request; if not, any subsequent validate due to a temporally silent store to the cache line on the owning processor is aborted. This mechanism eliminates many useless validates without sacrificing any opportunity; if no remote processor had a valid copy of the cache line at the intermediate value store, it is not possible for the cache line to be in T state, thus any validate is useless. The enabling mechanism for this optimization is the *shared* snoop-response, which is used in machines implementing E state. In this context we have overloaded its use slightly, as it is only used for Read transactions in conventional MESI. This snoop-response mechanism enables a very simple form of distributed decision-making among processors in the system in response to coherence requests since coherence actions taken by the requestor are affected by events in remote processors; in this case, valid data in remote caches.

Although assertion of the shared line is a fixed response in conventional protocols, it is easy to imagine using this distributed communication mechanism to implement predictive coherence mechanisms. For example, a remote processor may not assert the shared

signal in response to a remote Read request, even if it has a valid copy of the data. However, for correct protocol operation to result, it will need to treat the Read request as a ReadX, and invalidate its copy of the cache line, since the requestor may obtain the data in E state. In cases where the remote processor suspects migratory sharing may result, it can use this mechanism to communicate that fact to a remote processor implicitly.

We explore a predictive coherence mechanism for MESTI implemented using this mechanism which constitutes a near zero-state predictor.[17] We add a single stable state to MESTI, called *Validate_Shared* which is entered upon receipt of a validate in T state. Therefore, it is semantically equivalent to S state for local requests; the only modification is any local request transitions a Validate_Shared cache line to S state. Upon receipt of an external ReadX/Upgrade transaction, the behavior is as specified in MESTI, except the shared signal is not asserted; all other states/transactions behave as normal. We call this response the *useful snoop response* because assertion of the shared line on an intermediate value store indicates a previous validate was useful and that it prevented a remote miss. The MESTI state machine for this enhancement is shown in Figure 5-21.

This simple behavior change allows a distributed prediction mechanism for useful validates. If a remote processor has not accessed the location since the previous validate, and therefore it is in the Validate_Shared state, it will not send a shared response upon subsequent intermediate value stores. Elimination of the shared response indicates to the requestor that no remote copies were present. Hence it should abort future validates, as done in the Snoop-Aware Validate policy. If a remote processor has accessed the location,

17. We call this predictor “near zero-state” because it is created by adding a single stable state to the L2 coherence state machine, which may have a complexity cost, but most likely will not have any additional bit cost in the state machine.

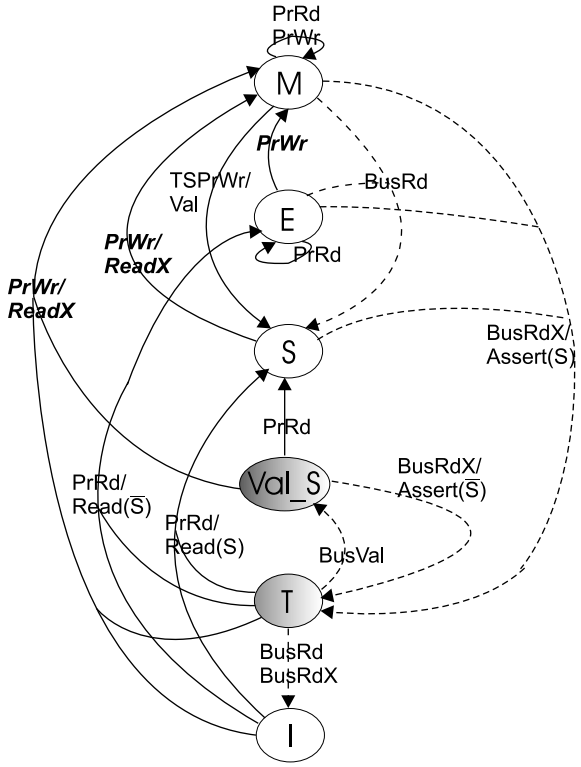

FIGURE 5-21. State Machine for Enhanced MESTI Protocol. We augment MESTI (Figure 5-6 on page 118) using the notation from Culler and Singh [31] to include support for the useful snoop response. Bus events which assert the shared line are indicated.

it will have transitioned to the Shared state, and will indicate a shared response, implying to the requestor that future validates may indeed be useful. Such a predictor allows detection of simple sharing scenarios in which a validate is never useful, but a remote cache maintains a valid copy of the data. This may occur during process migration; a process is moved for load-balancing purposes from CPU A to CPU B, however, its working set is entirely resident on CPU A. For large caches, much of the working set may stay cache resident on CPU A, thus temporally silent writes occurring to process-local data on CPU B may lead to address thrashing even with the simple Snoop-Aware Validate policy. This simple mechanism can avoid this, and similar, scenarios.

Figure 5-22 shows communication misses captured and additional address traffic present in the system with this simple enhancement. Results are normalized to the baseline

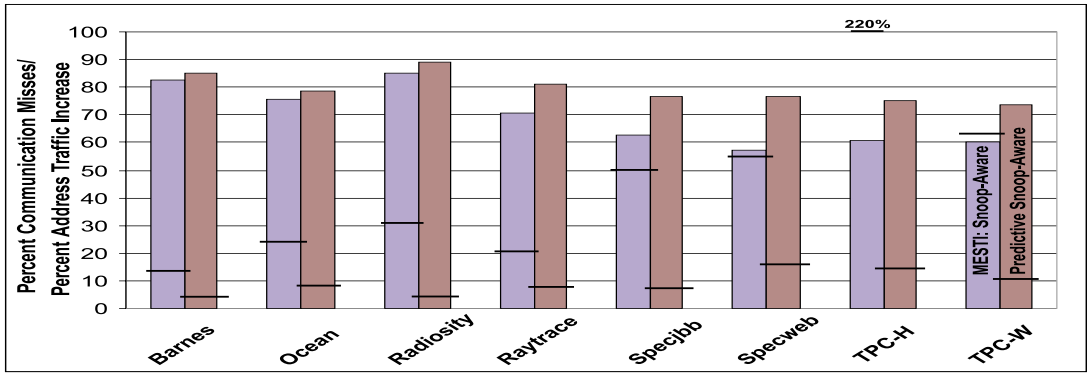

FIGURE 5-22. Predictive Snoop-Aware Validate Characterization Data. Results for MESTI (Snoop-Aware Validate policy) and the Predictive Snoop-Aware Validate policy are shown. Communication misses compared with the baseline are indicated with solid bars and address traffic increases are indicated with a line on top of each bar. Smaller values for the solid bars and lines on each bar indicate better performance.

and infinite caches, with MESTI results for comparison. In the scientific workloads, we observe significant reductions in address traffic with little opportunity in sharing reduction sacrificed. The only exception is raytrace, which sacrifices 30% of sharing miss opportunity (11% of all communication misses); however, additional address traffic is decreased 63%, which indicates a decent trade-off. In the commercial workloads, we observe a substantial address traffic reduction; all workloads show less than 17% increase over the baseline, with the additional traffic reduced by 87% on average as compared to basic MESTI. Most noteworthy, the previously observed address traffic increase in tpc-h (220%) has been reduced to less than 15%, with only 37% of the opportunity of communication miss reduction lost. However, substantial sharing reduction is lost in all commercial workloads, 37% on average. Therefore, while this optimization is effective, the simple prediction mechanism employed does not effectively capture all TSS sharing patterns. For example, a simple pattern which cannot be exploited for sharing miss reduction with this scheme is shown in Table 5-6.

Behavior under MESTI is shown in the top half of the table, predictive snoop-

Table 5-6: Illustration of Lost Opportunity with Predictive-Snoop Aware Validate. The top of the table illustrates behavior with MESTI including snoop-aware validate, the bottom illustrates behavior with predictive snoop-aware validate. Predictive snoop-aware validate disables the shared snoop-response at T3, aborting the validate at T4, and thus leading to the Read (miss) at T5 which is captured with basic MESTI.

Time	CPU 0			Snoop-Response	CPU 1		
	Instruction	Cmd/Txn	State		Instruction	Cmd/Txn	State
T0	LD [A] (0)	Read	Shared		LD [A] (0)	Read	Shared
T1	ST [A], 1	Upgrade	Modified	Shared			
T2	ST [A] , 0	Validate				Remote Validate	Shared
T3	ST [A], 1	Upgrade	Modified	Shared			
T4	ST [A], 0	Validate					
T5				S	**LD [A] (0)**	**<Hit>**	Shared
Predictive Snoop-Aware Validate Policy							
T0	LD [A] (0)	Read	Shared		LD [A] (0)	Read	Shared
T1	ST [A], 1	Upgrade	Modified	Shared			
T2	ST [A] , 0	Validate				Remote Validate	Validate_ Shared
T3	ST [A], 1	Upgrade	Modified	<None>			
T4	ST [A], 0						
T5				S	**LD [A] (0)**	**Read**	Shared

aware validate in the bottom. The key point is behavior at T3; in MESTI the shared snoop-response indicates a remote processor can benefit from temporal silence if detected, and thus leads to a validate at T4. In predictive snoop-aware validate, the addition of the Validate_Shared state disables the shared snoop-response at T3, thus no validate is broadcast at T4, leading to the Read (miss) for CPU 1 at T5. In short, this simple predictor only captures TSS sharing patterns where each validate is useful; a single useless validate will causes any subsequent temporal silence detection to avoid validation until a remote miss is observed. A potential method to eliminate this drawback is to add hysteresis at remote nodes; we do not explore this for the sake of brevity.

5.7.3 Memory Address-Based Prediction

Since the simple predictor described previously is not sufficient to capture more complicated sharing patterns, we also explore a memory address-based prediction mechanism which attempts to determine the temporal silence and sharing patterns a given cache

line exhibits. A block diagram of the predictor as well as a state diagram governing transi- 162
tions is shown in Figure 5-23.

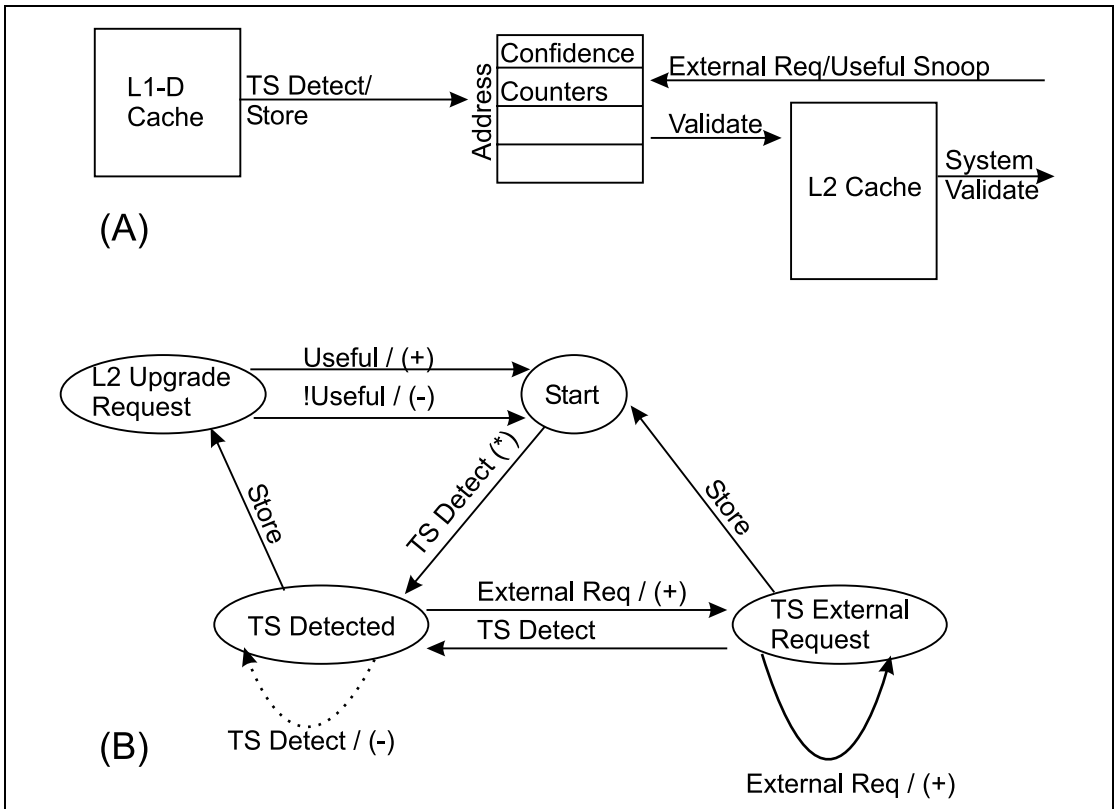

FIGURE 5-23. Address-Based Useful Validate Predictor. The predictor observes L1-data cache events and also system snoop requests/responses to determine when a validate broadcast in MESTI might prevent a remote cache miss. The system block diagram is shown in (A), while (B) indicates the Mealy state machine used to determine when the confidence counters for each cache line are updated. Transitions labeled (+) indicate confidence increments, (-) indicate confidence decrements, and (*) indicates the transition where the confidence value is read to determine system validate broadcast. The dashed transition and confidence decrement is needed for the Reupgrade policy discussed for predictors between the L1-D and L2 caches observing a filtered reference stream (Section 6.6.2).

The predictor observes temporal silence detection (TS Detect) and store requests from the L1-D cache and also external requests/responses from other processors in the system. On each L1-D detection of temporal silence, the predictor indexes into an array of confidence counters [102] to determine whether a validate should be sent to other processors in the system. If the confidence is above some threshold, a system validate is broadcast and the cache transitions to shared state; if below the threshold, a system validate is

not broadcast and the cache returns to exclusive state. This check is indicated by (*) in Figure 5-23.

The state machine governing confidence counter updates is shown in Figure 5-23(B). At a high level, the state machine attempts to predict critical vs. anti-critical temporal silence. It does this by transitioning to the TS Detected state whenever temporal silence occurs. If an external request occurs for the cache line while it is temporally silent this indicates useful temporal silence, and thus a confidence increment, as indicated with the External Req arc. When a subsequent non-update silent store occurs in the TS Detected state, the useful snoop response trains the predictor and detects anti-critical temporal silence; we discuss this in detail shortly.

The TS External Request state allows multiple confidence increments depending on the degree of sharing of the temporally silent cache line. Each external request to cache lines which have reverted causes a separate counter update. When another (non-update silent) store occurs, the state machine again waits for the occurrence of temporal silence.

A potential pitfall when designing this predictor is continually updating it once remote misses are being eliminated with validates sent to other processors; once misses are not observed (because of action taken to prevent them) information about usefulness of a validate may be lost. Put another way, it is easy to imagine training a sharing-predictor when misses to cache lines of interest are observed, but training a sharing-predictor once the misses are being eliminated is more difficult. In a straight-forward manner, the misses can be periodically re-introduced (by not sending validates for lines which exhibit temporal silence), to be sure remote processors are still benefiting from the validates, similar to the way hybrid update/invalidate protocols such as Dragon exit the update mode

[31]. However, we have already discussed the useful snoop-response in Section 5.7.2 which remote processors can send in response to intermediate value store upgrade requests. We show the use of this response to differentiate between useful and useless cases of temporal silence in the transition from the TS Detected to L2 Upgrade Request to Start transitions. Note that a separate state is needed for L2 Upgrade Request because the useful snoop response is generated significantly after the store arc; the coherence agent must collect all snoop responses to determine usefulness. Making each intermediate value store visible (Section 3.1) makes such continuous training possible.

The predictor and confidence counters can be implemented as a separate structure which covers a smaller working set than the L2 cache because prediction resources are only required for cache lines exhibiting temporal silence. However, implementing a separate tagged predictor will have significant area overhead for only the tags since each predictor entry is only 2 bits for the state machine in Figure 5-23 plus confidence counters.[18] Two methods can significantly reduce this overhead: making the predictor tag-less, at the cost of significant aliasing of a direct-mapped predictor or combining predictor entries with the existing L2 tag array. In many 2-level cache implementations, confidence counter access and predictor update can be performed only on L1-L2 cache transactions. For example, this is possible with an MSI L1-D cache MOESI L2 cache, which we use for performance studies in Chapter 6. Integrating these structures with pre-existing L2 cache tags is desirable to reduce area overhead. Further, accessing and training only on L1-L2 transactions greatly reduces the prediction bandwidth required since the reference stream observed is filtered by the L1 cache.

18. Next-state and output logic can be shared among all entries and therefore we neglect it in area estimation.

In this configuration, the storage overhead for the predictor is a function of L2 cache size and mapping; as an example, for a 16MB, 8-way associative, 64-byte line cache with 40 bit physical address the overhead is 5 bits (2 bits for predictor states plus 3 bits for confidence counters) / (~(6 baseline state bits) + (19 tag bits) for existing L2 cache tags) = a 20% increase in L2 cache tag storage to implement the predictor. Since cache tags are only a small fraction of the overall silicon area, the vast majority is the data array, the predictor storage overhead is likely less than 5%.

Furthermore, note that this prediction mechanism can be implemented entirely outside the processor core because it only observes physical memory addresses and standard coherence transactions; no PC or additional correlation information is used. The only event beyond standard coherence transactions required for the predictor is temporal silence detection. Slightly augmenting the L1-D cache to indicate temporal silence with perfect timeliness is advantageous. However, the predictor can be completely implemented at the L2 cache by simply waiting for L1-D cache writebacks (as discussed in Section 5.5.4), thereby avoiding L1-D cache modification.

Tuning the predictor cold miss confidence, confidence threshold, and confidence change values for various events was determined experimentally. We used a differential tuning algorithm which traded off observed address traffic increases against TSS misses eliminated for different predictor configurations. Studies not detailed here determined that broadcasting validates for predictor misses was detrimental in general; for large predictors these are largely cold misses. We show results for three of the most promising configurations (initial confidence: 3, confidence threshold: 4, increment: 2, decrement: 1, and saturation value: 7, i.e. 3-4-2-1-7; 3-4-1-1-7; and also 3-4-1-2-7) in sharing misses avoided

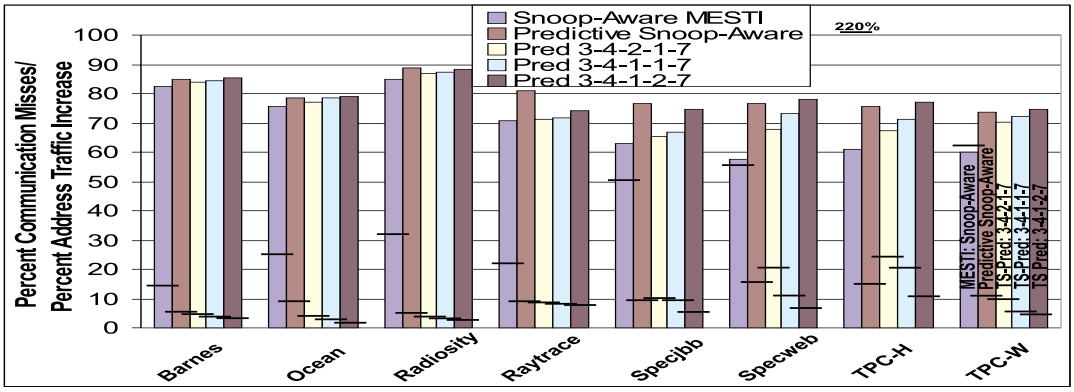

FIGURE 5-24. Address-Based Useful Validate Predictor Characterization Data. Results for MESTI (Snoop-Aware Validate policy), Predictive Snoop-Aware Validate (Section 5.7.2), and three different predictor configurations are indicated. Communication misses compared with the baseline are indicated with solid bars and address traffic increases are indicated with a line on top of each bar. Reduced communication misses along with lower address traffic indicates a desirable configuration.

and address traffic increase over the baseline case for infinite caches in Figure 5-24.

We observe that the predictor reduces anti-critical validates significantly as compared with simple snoop-aware validate and measurably over predictive snoop-aware validate. The predictor achieves measurable communication miss improvements while also reducing address traffic over predictive snoop-aware validate, thus eliminating the vast majority of additional address traffic in the scientific workloads while sacrificing little opportunity; however, in the commercial workloads the performance is more variable, although better than predictive snoop-aware validate in all cases. The predictor is least effective in *specweb*; although it decreases address traffic substantially, it cannot do so without losing substantial opportunity for sharing reduction. Figure 5-18 shows that the working set of cache lines exhibiting temporal silence is not large in *specweb*, indicating that this predictor cannot reliably distinguish between useful and useless temporally silent stores for the patterns exhibited in this workload. More elaborate prediction methods could improve this result; however we emphasize this prediction mechanism can be implemented entirely outside the processor core, as no program structure (PC values) or

other commonly used correlation information, e.g. branch histories, are utilized. Therefore, as described previously (Section 5.4.2 and throughout Section 5.5), a system can effectively implement MESTI for communication miss reduction with essentially no changes to the processor core itself, although additional benefit may be realized with such modifications.

This simple mechanism with low predictor storage overhead eliminates most anti-critical validates while still reducing communication misses substantially in commercial workloads. Reductions in communication misses of 32%, 30%, and 25% on average for 3-4-2-1-7, 3-4-1-1-7, and 3-4-1-2-7, respectively are observed. It achieves near-perfect results for the scientific workloads studied (except *raytrace*). We explore detailed performance results in Section 6.6.2.

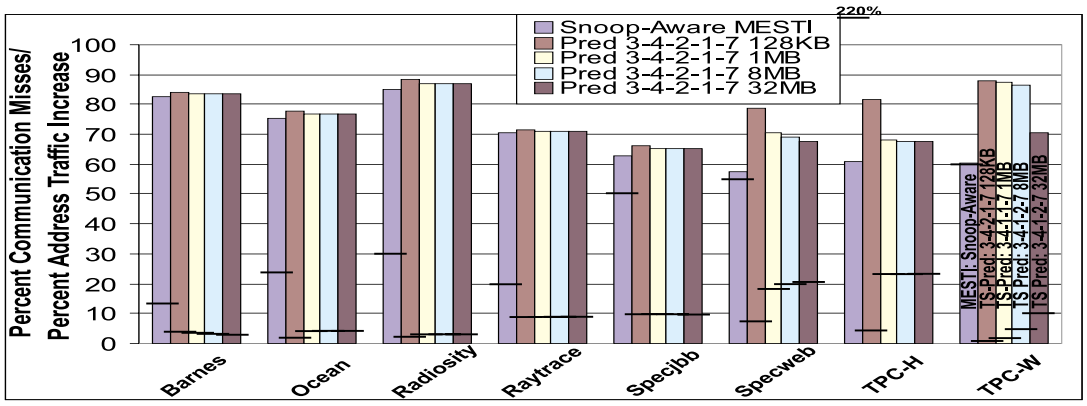

FIGURE 5-25. Characterization Data for Different Predictor Sizes. Results for MESTI (Snoop-Aware Validate policy), and four different predictor sizes are indicated. Communication misses compared with the baseline are indicated with solid bars and address traffic increases are indicated with a line on top of each bar. Note that the capacity indicated for each predictor is its effective cacheable area, not number of bits used in the predictor.

Although we have argued that significant silicon area for the predictor can be saved by combining it with the L2 cache tags, we show performance for our 3-4-2-1-7 predictor with varying cacheable space in Figure 5-25. Results for predictors capturing 128KB, 1MB, 8MB, and 32MB of L2 cache accesses are shown. Each predictor is 8-way

associative. Actual bit storage for each predictor can be calculated as illustrated previously; e.g., for 40-bit physical address and 64B cache lines the 128KB reach predictor requires: 5 bits (state machine/confidence) + 26 bits (tag) = 31 bits = ~4B per cache line of reach. This implies the 128KB reach predictor requires actual storage of ~8KB. Implementing a tag-less predictor can reduce the storage overhead substantially at the cost of increased aliasing.

In the scientific workloads, the smallest predictor is sufficient to capture all opportunity. This correlates well with data presented in Figure 5-18, indicating a small working set of TSS cache lines in these workloads. Results for the commercial workloads correlate similarly; however, in *tpc-w* we notice that even a large, 8MB-reach predictor sacrifices substantial opportunity even though the working set of TSS lines is less than 1MB (Figure 5-18). A similar trend is true in *specweb* and *tpc-h*; a larger predictor than the working set illustrated in Figure 5-18 is required. This occurs because predictor entries are allocated for each temporal silence detection, regardless of sharing patterns exhibited for the cache line. Therefore, extra predictor capacity is required to filter out cache lines which exhibit temporal silence but do not actively participate in sharing.

Finally, note that the smallest predictors, although they have the lowest coverage of sharing misses eliminated, tend to have the least address traffic increase as compared to the baseline case. This indicates address-based prediction is more accurate for small, but active, data working sets as compared with larger, less-active working sets.

5.7.4 Exploiting MESTI to Ease Handling of Update Silent Store Misses

At this point, we revisit a question we deferred from Section 4.3, namely, providing an efficient mechanism for handling update silent store misses. As explained there, we

desire a mechanism that allows sending a ReadX transaction for all store misses since the vast majority of store misses are not update silent, but can still exploit USS when store misses are update silent to prevent remote communication misses.

This problem is efficiently solved by simply treating store misses as candidates for exhibiting temporal silence, where the intermediate value store and the temporally silent store are the same dynamic store. In the case of a store miss, a ReadX transaction is sent, and if the store is update silent, it is immediately followed with a Validate transaction. In this way, store commit in the core is not stalled in the common case of non-update silent store misses, with validates in the case of update silent store misses allowing remote processors to eliminate communication misses. This scenario is equivalent to USS-P (assuming instantaneous address transactions) in terms of communication misses eliminated (presented in Section 4.2). Unnecessary address traffic is exactly equivalent to anti-critical update silent store misses (presented in Section 4.4).

Furthermore, the prediction mechanisms discussed in Section 5.7.2 and Section 5.7.3 can also be used unmodified. If an update silent store miss is encountered, and a validate is broadcast, the predictors monitor for a subsequent non-update silent store. The upgrade caused by this non-update silent store relies on the useful snoop response for training purposes. This allows elimination of future anti-critical update silent store misses to the same cache line. We use this mechanism (a ReadX/Validate pair) for handling update silent store misses in our performance studies in Chapter 6.

5.8 Memory Consistency and Correctness Implications

We discussed consistency and correctness in the context of update silent stores in Section 4.5. In this section, we discuss these issues for temporally silent stores in WC TSS

(Section 5.4.1) and the MESTI protocol (Section 5.4.2). Correctness under other methods of exploiting TSS, i.e. SLE (Section 5.4.3) and load value prediction (Section 5.4.4) have been discussed in other works [90] [76].

5.8.1 Memory Consistency Considerations

WC TSS is obviously an extension of delayed consistency, and inherits the same requirements as the proposals on that subject [34]. As a brief summary, WC TSS only allows collapsing temporally silent pairs that need not be made visible to remote processors—in weak consistency a simple condition for correctness is that only temporally silent pairs not separated by a memory barrier may be collapsed. Once a memory barrier is reached, all current values should be made visible and any opportunity for collapsing temporally silent pairs across the barrier must be forgone. This can be accomplished simply by allowing all current store values in write buffering structures to be propagated into the memory hierarchy.

MESTI (as compared with update silent store suppression in Section 4.5.1) is significantly less complicated to prove. In fact, no detailed argument is required. The key observation is that all stores present in the original execution are still visible; the temporally invalid state is considered equivalent to the invalid state. Therefore, as long as the MESTI protocol is correctly implemented and properly maintains coherence, no impact is observed regarding consistency.

5.8.2 Existing ISA Correctness Considerations

In Section 4.5.2 we discussed correctness considerations for update silent store suppression, which included side-effects through the reservation register in architectures utilizing load-locked/store-conditional synchronization primitives and also other side-

effects, e.g. page table references. Again, since MESTI makes both stores of the temporally silent pair visible, and neither of these stores is update silent, these problems are easily addressed.

As an example, we reconsider the issue with reservation granule side-effects in PowerPC introduced in Section 4.5.2 on page 98. Suppose, without loss of generality, CPU 1 performs a temporally silent pair of writes while CPU 0 is performing the detection algorithm outlined for update silent stores in Figure 4-8 on page 101, i.e. the temporally silent pair corresponds to the update silent store in the figure. Any temporally silent pair performed between the synchronizing constructs is observable since the intermediate value store behaves exactly as it would in the baseline implementation, clearing CPU 0's reservation. If the intermediate value store or temporally silent store falls outside the synchronizing construct, the intermediate value or the temporally silent value becomes explicitly visible through a ReadX/Upgrade, clearing the reservation and maintaining correct operation.

5.8.3 Correctly Implementing the MESTI Protocol

The statements throughout this section have assumed a correct MESTI coherence protocol when discussing both consistency and architectural observability. However, correctly implementing the MESTI protocol in practice requires special consideration. We provide additional comments on correctly implementing MESTI in our detailed simulation infrastructure (PHARMsim [15]) in Section A.1.

The MESTI protocol is trivially correct in the case where no validates are broadcast because it is functionally equivalent to MESI. The only concern with correctly implementing MESTI is ensuring that validates observed by remote processors in temporally

invalid (T) state correspond to the temporally silent value on the broadcasting processor. This means a validate may only be observed by remote processors if the T state corresponds to the intermediate value written by the validating processor. Furthermore, in any subtle race condition, it is always correct to eliminate or ignore a validate, and correct operation with forward progress will still result, regressing to the MESI protocol.

Practically speaking, correct correspondence can be assured through snooping any pending validate, e.g. the delay queue described in Section 5.7.1 or other outbound coherence queues, for internal and external coherence conflicts. Once a validate has left the processor, correct correspondence can be assured through canceling any validate not associated with the current owner when the validate reaches the coherence ordering point. In snoop-based protocols which we have assumed previously throughout this chapter, this is the address network. We discuss directory-based schemes separately in Section 5.9. Other more complex system topologies (hierarchical snooping or ring interconnects) may present additional complication, which we do not explore for the sake of brevity.

Another complication, in practice, is maintaining ownership of a line which has left Modified state because it has been validated, but which has not yet successfully transferred ownership of the cache line back to memory to source the data for subsequent requests. This is essentially the *writeback race* problem in MESI, which is well understood [31]; the same solutions can be applied here. The protocol used for performance evaluation is MOESI, where this race can be simply handled by transitioning to the Owned state when a validate is broadcast. In owned state the validating processor is still responsible for providing data to other snoop requests for validated lines.

Finally, correct temporal silence detection for the MESTI protocol must differenti-

ate between valid and invalid stale data. In the case of write, no-allocate operations where the stale data is not acquired before the write, data present in the cache may not be used for temporal silence detection, otherwise incorrect operation may result. Examples are the PowerPC *dcbz* (data cache block zero), Alpha write-hint, and similar cache control instructions which only obtain exclusive coherence permission without a data transfer. These cases can be avoided by adding a Modified_NoValidate state indicating that the current data is modified but is not a candidate for validation.

5.9 Extending Temporal Silence Exploitation to Directory-based Systems

We have described numerous communication mechanisms and argued about their correctness and suitability to snoop-based multiprocessor systems throughout this chapter. In this section, we offer insight on extending the proposed mechanisms and prediction techniques to directory-based systems (many examples of such systems are illustrated by Culler and Singh [31]). We do not offer rigorous performance evaluation or correctness arguments for such systems, because we have only validated our proposals in the PHARMsim environment which implements a snoop-based interconnect (see Section 6.2); rather we illuminate some fundamental issues which may present complication in common directory-based implementations.

5.9.1 Correctness Considerations

Considering the four temporal silence communication methods presented throughout previous sections: WC TSS (Section 5.4.1), MESTI (Section 5.4.2), SLE (Section 5.4.3), and LVP (Section 5.4.4), we reiterate that correctness requirements for WC TSS, SLE, and LVP for directory-based coherence schemes have been explained elsewhere, i.e. [34], [90], [76]. Very simply, since all mechanisms rely on implicit communication, exist-

ing coherence mechanisms can guarantee correctness. MESTI is an interesting case, and we discuss its correct implementation in directory-based systems currently.

We have already described the problem of correct correspondence between validate transactions and temporally invalid (T) states which is fundamental to ensuring proper implementation of MESTI in any system in Section 5.8.3. In directory-based systems, correct correspondence can be assured by verifying the proper pairing between a validate and an owning processor at the coherence point, which is the home node, and canceling any improper validate. However, this only assures correct correspondence on the path from the owning processor to the home node. Once proper validates leave the home node, correspondence must still be maintained from the home node to all targets of the validate. Ordered interconnects from the home node to other processors can assure correspondence; assuring correspondence in unordered interconnects either requires acknowledging validates (in effect, making the interconnect appear ordered) or more elaborate schemes.

Beside ordering requirements in the interconnect, we note that T state should be implemented at the directory itself (and not only in remote processor nodes) to ensure correctness. As discussed in Section 5.8.3, the MESTI protocol is trivially correct in the case of no validate broadcasts. However, MESTI relies on observation of other address transactions when in T state to help ensure correct correspondence on remote processors. If T state is not reflected at the directory, multiple address transactions may need to be broadcast on each remote processor request to ensure remote caches properly enter the invalid state and subsequently ignore non-correspondent validates. A simpler solution is to augment the sharing list to also reflect T state. With this support, it may be possible to only

send a single invalidate, and no additional address transactions, to processors in valid states and maintain correct operation. Although the single invalidate will cause remote processors to enter T state (potentially creating a correspondence hazard when subsequent address transactions are filtered by the directory), since T and I states are equivalent from the perspective of local processor requests, correct operation can still be maintained. The only requirement is for the directory to ensure no subsequent validate is sent to non-corresponding T state lines.

5.9.2 Feasibility of Performance Optimization Techniques for MESTI

We discussed various methods of improving performance of the basic MESTI protocol: snoop-aware validate (Section 5.7), validate delay queue (Section 5.7.1), predictive snoop-aware validate (Section 5.7.2), and memory address-based temporal silence prediction (Section 5.7.3). We now briefly discuss fundamental aspects of snoop-based systems which each might exploit and possible extensions to directory protocols. We point out that the validate delay queue is a processor-local optimization, hence we expect no interaction with the coherent interconnect type in its implementation and do not discuss it further here.

The snoop-aware validate policy relies on the same mechanisms used to implement the exclusive (E) state. Directory-based systems which can implement E state should also be able to implement the snoop-aware validate policy. Fundamentally, the requirement to enable snoop-aware validate is knowledge of emptiness/non-emptiness of the sharing list upon an upgrade/invalidate message. Principally, this means the directory must provide a response to the upgrade request with sharing information; since the directory normally provides some type of response to indicate ordering, augmenting this response

with sharing information may be possible. However, in some protocols, responses are generated before the directory is consulted; therefore this optimization may not be possible. For example, in the Alpha GS-320 [41] the response can be generated when the request has been accepted by the directory but has not yet interrogated the directory contents to obtain sharing information.

Additionally, predictive snoop-aware validate and memory address-based temporal silence prediction rely on the *useful snoop response* (Section 5.7.2) to enable a distributed communication mechanism indicating the usefulness of validates at remote processors. Fundamentally, obtaining information about usefulness of validates at remote processors requires the processors themselves to respond to each invalidate/upgrade request. In cases where only the directory responds to the requestor, this information may not be available. However, a potential solution is to have each processor which accesses a Validate_Shared block send a message to the directory indicating validate usefulness. This message need not delay local access to the Validate_Shared block (it can still behave as a cache hit) since the message is only used to indicate validate usefulness. Also, we expect additional training messages to be minimal, equivalent to transactions observed in the baseline case without validate messages. Essentially, this mechanism implies each MESTI avoided miss still creates a transaction to the directory (so address transactions are equivalent to the baseline case), however data transactions and associated latency can still be avoided. Finally, we can implement predictors which do not rely on the useful snoop response to aid training. We have discussed this in Section 5.7.2.

5.10 Program Behavior

We have described many aspects of temporally silent program behavior directly

and indirectly throughout previous sections. For example, Figure 5-10 indicated that temporally silent pair distance can be substantial in commercial workloads; Figure 5-18 showed the dynamic memory footprint of cache lines exhibiting TSS can be widely variable between workloads; and Figure 5-20 illustrated distance in processor cycles between useless temporally silent writes and the subsequent write, as well as temporally silent last-write to remote access distance to show the effect of delaying temporal silence detection. In this section we perform additional examination of the TSS phenomenon, not focused on deriving efficient mechanisms to capture it, but for the sake of TSS itself. We explore contribution to TSS misses from cache lines accessed with explicit atomic memory operations, value distributions for intermediate value and temporally silent stores, and conclude with an examination of operating system, library, and user code exhibiting TSS.

5.10.1 TSS Contributed by Explicit Atomic Operations

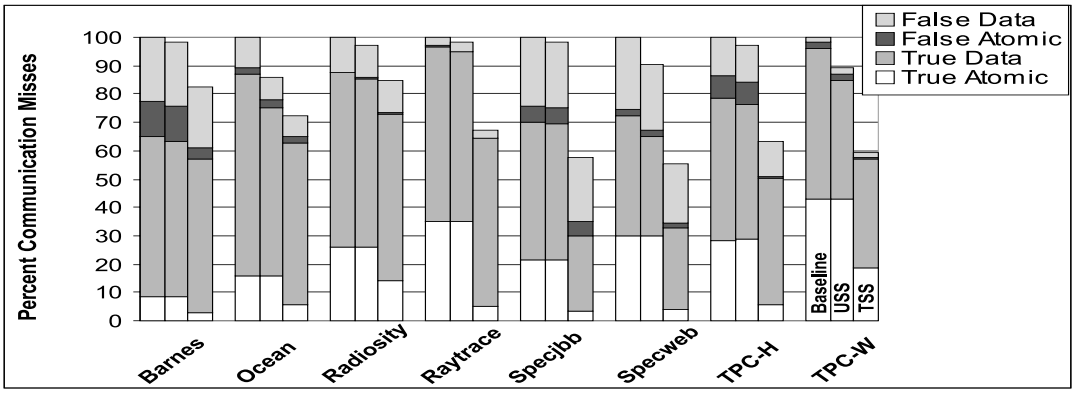

FIGURE 5-26. Contribution of Atomic Operations Under Different Definitions of Sharing. The figure indicates the contributions of cache lines written with at least one stwcx/stdcx (store-conditional) operation to sharing misses for Baseline, USS, and TSS. The data is normalized to sharing misses in the Baseline case.

Lock variables are great candidates for exhibiting temporal silence since they revert to their unheld value when released. Indeed, Speculative Lock Elision exploits "silent store-pairs" (which we refer to as atomic temporally silent store pairs) to elide

transfers and execute critical sections concurrently [93]. To further understand program behavior exhibiting TSS, we examined the percentage of sharing misses contributed from memory locations written with explicit ISA atomic operations—store conditionals for PowerPC—and those written with data operations.

In Figure 5-26 we show the contribution to sharing misses by atomic operations for Baseline, USS, and TSS. We determine explicit atomic operations in a very liberal manner: if any location within a cache line has been written with a PowerPC stwcx/stdcx (store-conditional) instruction, subsequent misses to that cache line are denoted as related to an explicit atomic operation. In reality, there may also be data operations within the same line. Load-linked/store-conditional instructions can implement various atomic primitives, including locks, atomic updates, compare-and-swap, atomic list insertion/deletion, and others [31]. Determining which atomic primitive is implemented is non-trivial without instrumenting the binary at compile time. Since most temporally silent pairs occur within the AIX kernel and within the various libraries which make up our commercial workloads, we are unable to further classify atomic primitives; we present additional data on classification in Section 5.10.4.

Examining the figure, we see that a large fraction of true sharing misses are due to atomic primitives across all benchmarks in the Baseline case, ranging from 9% in *barnes* to 43% in *tpc-w*. We also observe that over 75% of TSS avoidable misses are from atomic primitives except in *specjbb*, where the fraction contributed by atomic primitives is 45%. This data indicates we may be able to leverage explicit atomic operations to more efficiently exploit temporal silence. However, in *specjbb*, a majority of TSS avoidable misses are not due to atomic primitives, indicating a general mechanism—one not focused on just

these constructs—is desirable. Note that all mechanisms presented throughout previous sections do not differentiate atomic primitives versus data operations and are examples of general mechanisms.

5.10.2 Why Temporal Silence Isn't Only Due to Locks

We have described general mechanisms for achieving benefit from TSS in this thesis, not focused on any particular programming idiom. However, designing such general mechanisms may be misguided if they are unnecessary; as discussed in Section 5.4.3, Speculative Lock Elision (SLE) exploits temporal silence of lock variables and eliminates many TSS misses. If all temporal silence is only due to lock variables, mechanisms targeting such idioms may lead to substantially simplified designs. We have provided much characterizing data throughout this chapter which sheds light on this point, we now examine a few key pieces of it to reveal that all TSS is not simply due to locks.

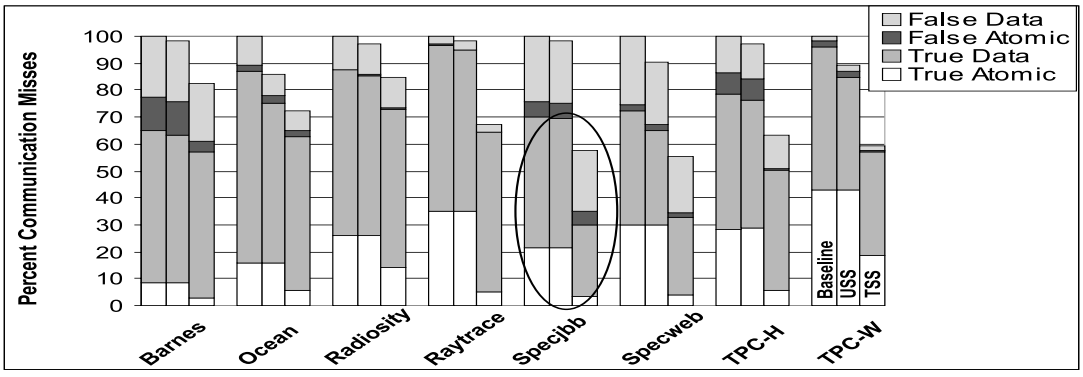

FIGURE 5-27. Contribution of Atomic Operations Under Different Definitions of Sharing. The figure indicates the contributions of cache lines written with at least one stwcx/stdcx (store-conditional) operation to sharing misses for Baseline, USS, and TSS. The data is normalized to sharing misses in the Baseline case.

In Figure 5-27, we reproduce the data presented in Figure 5-26 indicating the fraction of communication misses due to detectable synchronization references. As discussed in Section 5.10.1, non-trivial fractions of data communication misses can be eliminated

under TSS across all benchmarks. However, we explicitly note *specjbb* in Figure 5-27; 55% of communication misses in this workload are due to data references, not detectable synchronization primitives. We also note that many constructs can be implemented with load-locked and store-conditional instructions; it may be reasonable to assume that most are atomic primitives of some sort, but they may not all be locks. As discussed in Section 5.10.1, in PowerPC load-locked and store-conditional instructions can implement many constructs, e.g. atomic list insertion.

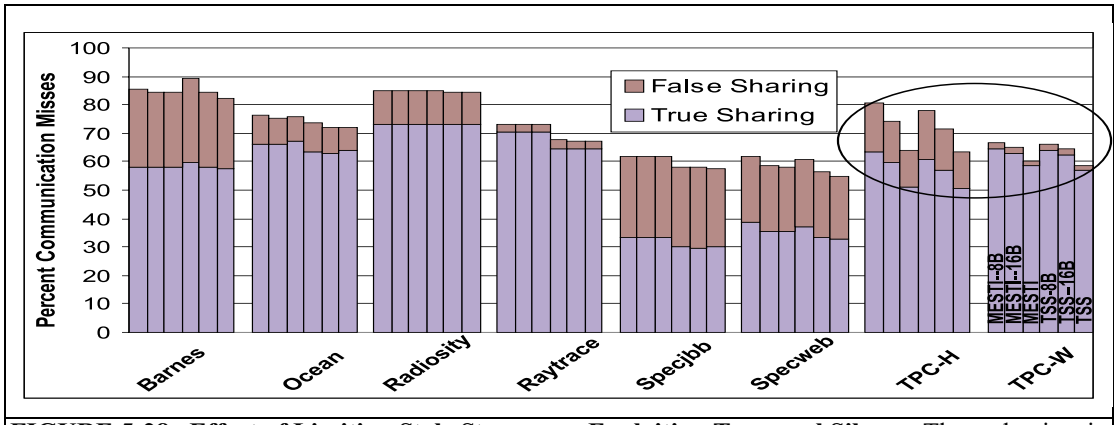

FIGURE 5-28. Effect of Limiting Stale Storage on Exploiting Temporal Silence. The reduction in communication misses normalized to the Baseline case (see Figure 5-7) for 64B cache lines and infinite caches for MESTI and TSS with limited stale storage (16B/cache line, 8B/cache line) is shown.

In Figure 5-28, we reproduce Figure 5-9 on page 131, indicating TSS captured with varying amounts of stale storage per cache line. In general, programming guides encourage allocating lock variables and data within critical sections to different cache lines to achieve improved critical section performance in the case of contention [31]; the same allocation is encouraged in the PowerPC architecture [78]. When this allocation is followed, if all TSS is due to locks, we expect no sensitivity to TSS captured for varying stale storage per cache line; as long as sufficient stale storage is provided for a single lock variable (maximally 8 bytes in PowerPC), all TSS should be captured. As shown in Figure 5-28, a substantial fraction of TSS in both *tpc-h* and *tpc-w* requires more than 8

bytes of stale storage per cache line, implying substantial TSS outside lock variables themselves.

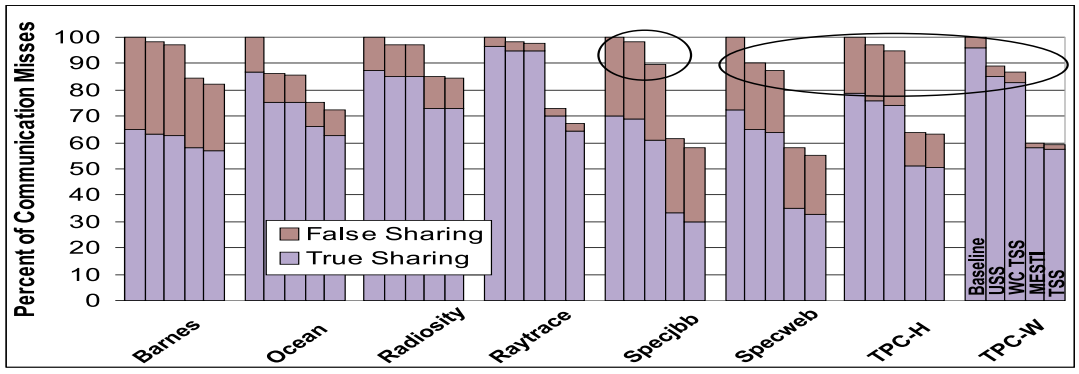

FIGURE 5-29. Percentage of Communication Misses for MESTI. The data is normalized to the Baseline case for 64B cache lines, including the reduction in communication misses under USS, WC TSS, and TSS for comparison purposes.

In Figure 5-29, we reproduce data from Figure 5-7 on page 120, indicating communication miss reduction for various non-speculative methods of capturing TSS (WC TSS, Section 5.4.1, and MESTI, Section 5.4.2). As discussed in Section 5.4.1, for a lock to implement its intended function under PowerPC, a memory barrier must occur between the lock acquire and release operations creating the temporally silent pair. WC TSS fundamentally cannot capture such temporally silent pairs due to the memory barrier. Figure 5-29 show that 3% to 9% of TSS misses can be captured by WC TSS; again indicating non-trivial TSS outside of lock variables.

We provide additional discussion of temporally silent program behavior through function-level examination in Section 5.10.4. However, we have shown through previous characterization that only targeting lock variables, or other synchronization primitives, is misguided; a robust mechanism for capturing TSS, such as those presented in this thesis, should be used to capture all available opportunity. Furthermore, we expect that a general mechanism, focused on exploiting TSS in-and-of-itself (and not due to a particular pro-

gramming idiom), will provide performance stability across diverse application classes and programming constructs.

5.10.3 Intermediate and Temporally Silent Store Value Distributions

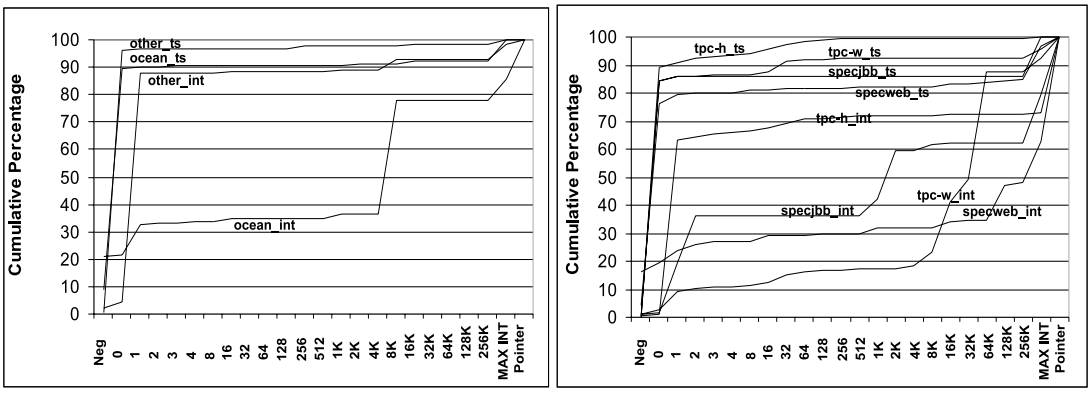

FIGURE 5-30. Cumulative Value Distributions for Useful TSS. The figure indicates the cumulative distribution of observed intermediate values and temporally silent values for TSS avoidable misses with scientific workloads shown on the left, commercial workloads on the right. Only values for integer stores are shown—contributions from floating point stores were also measured and negligible. Results for *barnes, radiosity,* and *raytrace* do not differ materially and are represented by the "other_int" and "other_ts" categories in the scientific workload graph.

The values observed for both intermediate values and temporally silent values of memory words may reveal insight into temporal silence. In Figure 5-30, we show the cumulative distribution of temporally silent values and intermediate values for TSS avoidable misses. Across both the scientific and commercial applications, the graph indicates over 75% of temporally silent values are zero, which is not surprising given similar results were observed for update silent stores [10]. However, in the commercial workloads, an observable fraction (over 5% in *tpc-w* and *specweb*) of temporally silent values are non-null pointers[19], a point which we will return to shortly.

Examining the intermediate value distributions for the scientific applications, we

19. We approximated pointer values by storing all virtual addresses touched by each process and assuming any observed value which matched a previously observed virtual address was in fact a pointer.

see (with the exception of *ocean*) the predominant intermediate value is integer one, which is caused by user-level spin-locks and other flag values. In *ocean*, the largest contribution comes from values in the range 4K-8K, which is unexpected. We examined this benchmark further, and found these values are actually the thread IDs used by AIX for the concurrent threads of *ocean*. These intermediate values still appear to be lock-related, with AIX using the thread ID to indicate which thread is holding the lock to thread-safe memory allocation routines and other operating system structures protected with the *simple_lock(), simple_unlock(), disable_lock(),* and *unlock_enable()* kernel routines. These are higher level locks (not simple spin-locks) provided by the AIX kernel [29]. In the commercial applications, the intermediate value distributions show strong contributions throughout the range, many of which match thread IDs of running processes (in the range 512-64K). This enlarged contribution from high-level locks in the commercial applications is reasonable; under a highly concurrent commercial application load we expect fewer simple spin-locks because they can reduce system throughput by wasting processor cycles spinning. More noteworthy is a particularly strong contribution for intermediate pointer values. The contribution of these values is over 40% in *specweb* and 20% in *specjbb*. Many of these revert to null (and therefore are counted under temporally silent value zero), but some also revert to non-null values, indicating that temporally silent pairs also occur in what are likely shared data structures.

5.10.4 Temporally Silent Program Behavior

Table 5-7: Functions Actively Participating in Temporal Silence. The table indicates the percentage of dynamic intermediate value and reversion stores contributed within the specified functions in the benchmarks indicated for cases of useful TSS. (K) Denotes kernel functions, (L) denotes library functions, and (U) denotes user code. (*) Includes contributions from multiple functions listed individually within the table as well, so for *ocean* this column adds to more than 100%.

Benchmark	% TSS Misses	% Dynamic TSS Stores	Function	Comments
specjbb	40.0%	13.9%	check_lock (K)	Compare and swap with import fence (lock acquire) primitive
specjbb	36.5%	38.5%	libjava.a (L)	Java Runtime Environment
specjbb	22.4%	8.9%	libpthreads.a (L)	Thread management
specjbb	17.7%	8.5%	clear_lock (K)	Atomic write with export fence (lock release) primitive
specjbb	16.9%	18.5%	libjitc.a (L)	Java Runtime Environment—JIT
specjbb	0.0%	0.0%	all user code (U)	All user-level application code
specweb	40.2%	20.9%	rsimple_lock (K)	Recursive simple lock acquire
specweb	20.1%	3.8%	simple_lock (K)	Non-recursive simple lock acquire
specweb	20.0%	4.1%	simple_unlock (K)	Simple lock and recursive simple lock release
specweb	19.4%	1.9%	p_inspte_p64 (K)	Process creation/deletion, page table entry insert
specweb	14.7%	1.2%	v_inspft (K)	AIX kernel
specweb	13.7%	1.5%	v_lookup (K)	AIX kernel
specweb	13.2%	1.3%	delall_pte_p64 (K)	Process creation/deletion, page table entry delete
specweb	11.9%	2.6%	invtlb_ppc (K)	Process creation/deletion TLB manipulation/invalidation
specweb	11.8%	1.0%	v_delpft (K)	AIX kernel
specweb	6.2%	0.9%	px_rename_p64 (K)	AIX kerne)
specweb	5.9%	1.0%	rsimple_unlock (K)	Recursive simple lock release
specweb	4.8%	0.6%	v_scoreboard (K)	AIX kernel, includes v_descoreboard
specweb	3.0%	0.4%	insque/remque (K)	Shared queue management
specweb	0.3%	0.1%	all user code (U)	All user-level application code
ocean	70.6%	(*)53.7%	all kernel locks (K)	All kernel level locks/releases, not within application code
ocean	46.8%	14.9%	simple_lock (K)	Kernel lock acquires, not within application code
ocean	42.6%	13.2%	simple_unlock (K)	Kernel lock releases, not within application code
ocean	15.2%	14.0%	rsimple_lock (K)	Kernel lock acquires, not within application code
ocean	11.5%	1.9%	unlock_enable (K)	Kernel lock releases, not within application code
ocean	9.0%	1.4%	test_and_set (K)	Other atomic primitives, not called directly by application code
ocean	7.7%	1.9%	cs (K)	Atomic compare and swap (application code lock acquire)
ocean	7.6%	1.8%	user mode store (U)	Application code lock release
ocean	7.0%	4.9%	state_save_point (K)	AIX kernel
ocean	5.6%	3.4%	v_lookup (K)	AIX kernel
ocean	5.5%	1.6%	v_inspft (K)	AIX kernel
ocean	5.3%	3.1%	p_inspte_p64 (K)	AIX kernel
ocean	4.5%	3.5%	call_dispatch (K)	Kernel thread dispatching/scheduling
ocean	4.4%	1.7%	set_curthread (K)	Kernel thread dispatching/scheduling
ocean	4.2%	1.0%	other user code (U)	All user-level application code (not lock releases)
barnes	80.9%	6.8%	cs (K)	Atomic compare and swap (application code lock acquire)
barnes	80.0%	6.9%	user mode store (U)	Application code lock release
barnes	9.8%	4.3%	Kernel Locks (K)	simple_lock(), simple_unlock(), disable_lock(), unlock_enable(); not within application code
barnes	4.7%	3.0%	Other kernel (K)	Process creation/deletion and thread management
barnes	5.7%	78.6%	other user code (U)	All user-level application code (not lock releases)

In the previous sections, we illustrated contribution to TSS through explicit atomic primitives and gained additional insight through examining values of intermediate and temporally silent stores. We conclude our examination of program attributes by examining contribution to TSS at the source/function level. We perform the study by monitoring static instructions contributing to each dynamic instance of TSS and tabulating the contribution from each function to both dynamic temporally silent stores as well as dynamic TSS misses.

In Table 5-7, we show the contribution of different functions within user level code, shared library code, and kernel code. The "% TSS Misses" column indicates the percentage of TSS avoidable misses the given function participates in by contributing an intermediate value store or temporally silent store. Percentages within this column may add to greater than 100% because multiple functions may participate within a single TSS avoidable miss; for example: if a separate function is used for lock acquire and release, a single miss due to this temporally silent pair will be counted in this column for both the acquire function and release function.

The "% Dynamic TSS Stores" column indicates the percentage of all dynamic stores (either intermediate value store or temporally silent store) contributing to TSS avoidable misses that occur within the given function. Because this column is normalized to the total number of dynamic stores contributing to TSS avoidable misses, this column will add up to 100% (or less). However, the total is always less than 100% when functions are considered individually because it is only practical to present data for a subset of functions.

Because we do not have source code for the shared libraries, commercial workload

application code, or the AIX v4.3.1 kernel, it is difficult to discern exactly which program semantics are causing temporal silence to occur. However, when possible, we provide function names to allow reasonable conjectures to be made.

Focusing first on the scientific workloads, it is interesting that most locking-related TSS avoidable misses in *ocean* are not within the application itself, but rather, in kernel support routines. The contribution of application spin-lock acquires/releases is less than 8%. In *barnes* (representative of *radiosity* and *raytrace*), the contribution of user-level spin locks is large (over 80%). However, substantial contributions within the AIX kernel (14.5%) are still noted. Furthermore, user-level code not directly related to atomic primitives contributes less than 5% and 6% in each benchmark. This data indicates that studying temporal silence without considering operating system and library code, even for scientific workloads, may ignore substantial opportunity.

In the commercial workloads, most TSS avoidable misses in *specjbb* occur within the Java runtime environment or JRE (*libjava.a* and *libjitc.a*). As shown in Figure 5-26, it appears that non-trivial sharing misses related to atomic as well as data operations occur within the JRE. In *specweb*, many temporally silent stores occur within process management, TLB, and page table-related code. This seems reasonable given the structure of this implementation of *specweb* which spawns a new perl process for each incoming web request, causing frequent process creation and destruction [18]. Results for *tpc-w* were qualitatively similar to *specjbb* and *specweb* and are omitted for brevity.

We also note that the two metrics presented (% TSS Misses and % Dynamic TSS Stores) do not always correlate strongly with one another. In some cases, many dynamic stores are contributing to relatively few TSS avoided misses, while in other cases the con-

verse is true (most notably *barnes* in “other user code” and “cs”, respectively). Therefore, if our goal is eliminating communication misses, it may not be prudent to develop mechanisms which target each dynamic temporally silent store with equal effort and assume that a corresponding reduction in TSS avoidable misses will occur.

Finally, note that even for TSS avoidable misses which can be determined to be locks with high likelihood (see Table 5-7, Figure 5-26, Figure 5-30), the value distributions in Figure 5-30 indicate the transfers avoided predominantly indicate the lock is free. This implies that eliminating these misses improves synchronization performance, as we will examine in detail in Chapter 6.

5.11 Characterization Data for Larger Multiprocessors

We have presented detailed characterization data throughout Chapter 4 and Chapter 5 for 4-processor systems to limit simulation time and also make the number of runs tractable. However, it is important to understand whether the fundamental phenomena studied throughout these sections (USS and TSS) are prevalent in larger multiprocessors as well. Therefore, we present both Figure 5-31 and Figure 5-32 to indicate the prevalence of USS and TSS for a 16-processor system; the configuration is described in detail in Section 3.6 on page 71, and is similar to the 4-processor studies carried out previously.[20]

Figure 5-31 shows the percentage of cache misses (infinite caches, 64B lines) for the Baseline, USS, MESTI, and TSS configurations. As compared to Figure 5-1 on page 109, we observe a relatively larger fraction of cold misses in the 16-processor case, which is expected. In order to keep simulation time tractable, we must run fewer instructions per

20. The most notable difference is the absence of the *specweb* and *tpc-w* benchmarks; scalability issues in the benchmarks, largely caused by a slightly antiquated AIX kernel unable to run more modern Java Virtual Machines make 16-processor results uninteresting.

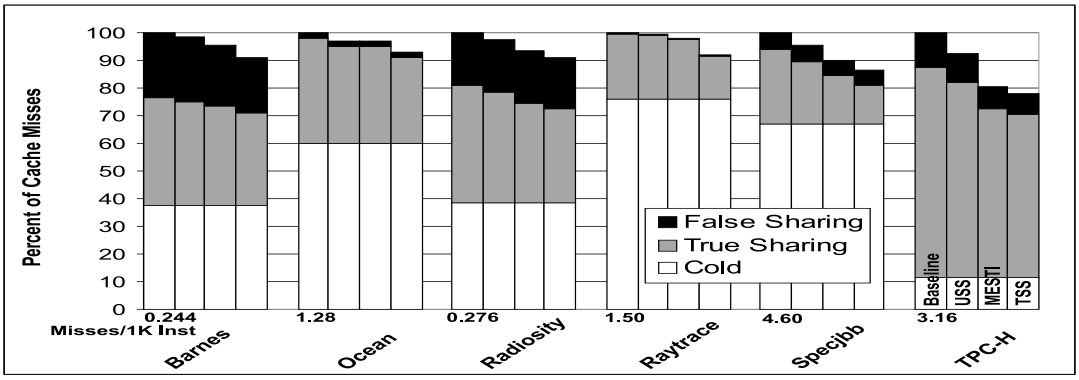

FIGURE 5-31. Percentage of Cache Misses for Different Definitions of Sharing. The data is normalized to the baseline for 64B cache lines and infinite caches for 16-processor benchmarks.

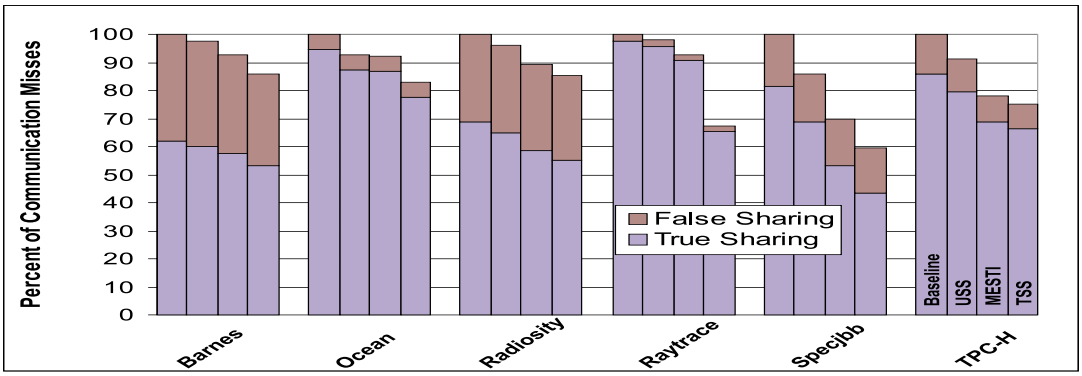

FIGURE 5-32. Percentage of Communication Misses for Different Definitions. The data is normalized to the baseline for 64B cache lines for 16-processor benchmarks.

processor for larger multiprocessors, implying a more substantial contribution from startup effects. *Tpc-h* is an exception, as the fraction of cold misses is actually reduced—additional communication misses introduced because of additional parallel processing elements counteracts the startup effects, as evidenced by the substantially increased miss-rate per instruction over the 4-processor case (Figure 5-1 on page 109). Figure 5-32 shows only the communication miss component. We observe that a non-trivial fraction of communication misses can be eliminated both under USS and TSS, similar to previous 4-processor data. However, we point out that MESTI is relatively less effective at approaching the TSS limit as compared to Figure 5-7 on page 120 for the 4-processor studies. This is intuitive; as illustrated in Section 5.4.2, MESTI only allows capturing a single previous

version for TSS detection and reversion. As processors are added, the likelihood of remote accesses while a memory location is in its intermediate value increases, and thus opportunity may be lost.

However, we emphasize that USS, MESTI, and TSS can still eliminate a substantial fraction of communication misses in 16-processor systems.

5.12 Related Work

5.12.1 Temporal Silence Exploitation in Multiprocessors/Multithreaded Architectures

We have discussed work from Rajwar and Goodman on Speculative Lock Elision (SLE) at length in Section 5.4.3. Follow-on work exploiting "silent store-pairs" is Transactional Lock Removal (TLR) which can re-order conflicting critical sections in the memory system to avoid many restarts present under SLE. Both are explained in detail in Rajwar's thesis [90]. A related proposal which builds upon thread-level speculation (TLS) hardware to implement speculative synchronization is presented by Martinez et al. [77]. However, this work does not exploit temporally silent stores but relies on instrumented binaries.

Other TLS studies have explored/exploited temporal silence, but under a different guise; proposals by Steffan et al. [103] and Cintra et al. [24] have shown the utility of both update silent stores and load value prediction to eliminate inter-thread dependences. Since both of these schemes use a last value predictor, they are effectively exploiting temporal silence on any successful load value prediction which eliminates an inter-thread dependence.

Finally, Dynamic Multithreading (DMT) exploits temporal silence of callee-saved registers to reduce inter-thread register dependence violations across function call-

spawned implicit threads [5].

5.12.2 Sharing/Coherence Prediction in Multiprocessors

We have devised coherence prediction mechanisms tailored to the MESTI protocol in Section 5.7. A plethora of related work exists in the realm of coherence prediction. Our proposals bear the greatest similarity to the work of Kaxiras et al. [53] and Martin et al. [74] in that they are targeted toward snoop-based multiprocessor systems. Kaxiras et al. focus on scientific workloads and advocate instruction-based prediction, while Martin et al. explore commercial workloads and illustrate the advantages of memory address-based prediction in such workloads. As we have pointed out extensively in this thesis, and as other researchers and practitioners have noted, sharing patterns, and therefore prediction mechanisms, for each workload class can vary significantly; therefore this thesis contributes substantially new material in this area by exploring both workload types simultaneously and devising mechanisms robust across workload types. Our temporal silence predictors are most similar to the migratory sharing predictor proposed by Kaxiras et al. [53] since they attempt to detect subsequent writes after a useless validate and broadcast-if-shared by Martin et al. [74] to detect actively shared data for validation. Martin et al. [74] as well as Nilsson et al. [87], [88] are the only prior works of which we are aware that study commercial workloads.

In the context of directory-based multiprocessors, last-touch predictors as proposed by Lai et al. [60] and dynamic-self-invalidation as proposed by Lebeck et al. [64] bear the greatest similarity to our work. Both proposals predict the last write to a cache block in order to relinquish ownership early and reduce 3-hop misses; a useful validate predictor attempts to achieve the same goal with the added benefit of also eliminating data

transfers. Additional related material can be found in other works [59] [86] [2] [45].

5.12.3 Update Coherence Protocols

The MESTI protocol bears similarity to update protocols which attempt to optimize multiprocessor performance by broadcasting writes to widely shared data. Updates can decrease latency experienced by remote processors for shared data, at the cost of additional address and data traffic. MESTI avoids additional data traffic by exploiting store value locality and requires less address bandwidth because only temporally silent data is validated, exploiting write combining; many hybrid protocols, e.g. Dragon and derivatives [31] [44] [20], can perform similarly, but do not exploit store value locality.

MESTI also inherits a drawback in common with update protocols; namely difficulty in providing write atomicity. Fortunately, because MESTI makes all intermediate values visible, the drawback is less apparent in MESTI; as long as the interconnection network can reliably determine if a validate is correctly associated with the owning cache, write atomicity can be maintained (explained in detail in Section 5.8.3). More elaborate schemes can be used in update protocols as well, however more often relaxed memory consistency constraints are exploited to circumvent this issue. As explained previously, MESTI has its heritage in invalidation-based protocols which have shown to be more desirable in commercial systems for many reasons. A discussion of trade-offs between these two major protocol types can be found in Culler and Singh [31].

5.12.4 Writeback Optimization and Memory System Optimization

Exploiting inclusion between the L1 and L2 by comparing the data value held in the L1-D cache against the L2 data is similar to eager-writeback proposed by Lee et al. [65]; using different LRU status to gate the comparison is similar to their eager-writeback-

queue. Using cache reference timing (similar to LRU status) to predict liveness of a cache block has been explored extensively in uniprocessors to optimize memory system performance by Hu et al. [50] through timekeeping within the memory system. Proposals to reduce cache power which determine the working-set size through tracking data referenced within a given time period, e.g. Kaxiras et al. [54], exploit similar behavior.

Chapter 6

Multiprocessor Performance Evaluation

Throughout previous chapters, we have presented characterization data on address traffic, data traffic, and cache miss rates to illustrate the potential benefit of exploiting store value locality in multiprocessor systems. In this section, we perform a detailed performance evaluation of the best schemes determined through our previous analysis in a full-system, execution-driven, integrated function/timing simulation environment. This simulation environment, PHARMsim, has been described elsewhere [15].

6.1 Motivation and Background

Industrial and academic practitioners alike have encouraged evaluation of architectural ideas with detailed performance simulators and quantitative approaches (most notably Hennessy and Patterson [46]). Furthermore, errors in simulation accuracy can exceed 100% [71] when the operating system is neglected, even for SPEC benchmarks; therefore, full-system simulation is necessary to ensure all relevant machine and program interactions are captured. Many academic projects (most notably, SimOS [99]) have extolled the virtues of full-system simulation for all architectural studies. Despite this, the de facto standard for most academic studies of single-threaded, as well as multi-threaded scientific workloads, have relied on user-code or user/library-code only simulation [14] [92] [89]. Since commercial workloads spend a large fraction of execution time in the operating system our studies have always focused on a full-system simulation environment [9, 95, 58, 6, 7]. In previous sections, we used a simplified processor model to make simulation time and design space exploration tractable.

To further prove the merit of this research, we now integrate the most promising ideas into a detailed, fully-integrated, full-system, execution-driven, simulation environment. PHARMsim has been described in detail elsewhere [15], however, we provide a brief overview of the simulator in this section.

6.2.1 The PHARMsim Environment and Its Heritage

PHARMsim is a PowerPC-based simulation infrastructure which uses large parts of the SimOS-PPC and SimpleMP simulators. SimOS is a complete machine simulation environment consisting of simulators for the major components of a computer system (cpus, memory hierarchy, disks, console, ethernet) [99]. We use a version of SimOS which simulates PowerPC-based computer systems running the AIX 4.3 operating system [56]. SimpleMP is a detailed execution-driven multiprocessor simulator that simulates out-of-order processor cores, including branch prediction, speculative execution and a cache coherent memory system [92] using a Sun Gigaplane-XB-like coherence protocol [22]. Integrating the SimpleMP simulator into SimOS required significant changes to SimpleMP in order to accurately support the PowerPC architecture; we discuss major modifications briefly for the interested reader.

SimpleMP was missing much of the functionality necessary to support system level code, in both the processor core and memory system. We augmented SimpleMP with support for all of the instructions (system-mode and user-mode) in the PowerPC instruction-set architecture. For some of the relatively complex PowerPC instructions, e.g. load/store string instructions, we use an instruction-cracking scheme similar to that used in the POWER4 processor which translates a PowerPC instruction into several simpler

RISC-like operations [105]. We also augment the processor core with support for precise interrupt handling and PowerPC context-synchronizing instructions, e.g., isync, rfi.

The SimpleMP memory system required major changes in order to support unaligned memory references, PowerPC address translation, and the set of PowerPC cache management instructions. To handle unaligned memory references, as allowed in the PowerPC architecture, the processor core splits each unaligned memory reference that crosses a cache block boundary into two smaller aligned references which are then individually issued to the SimpleMP memory system.

In order to accurately model PowerPC virtual memory hardware, we were forced to implement a PowerPC memory management unit (MMU) from scratch, including a translation lookaside buffer (TLB), TLB refill mechanism, and reference and change bit setting hardware. On a TLB miss, we simulate a hardware TLB miss handler which walks the page table by issuing memory references to the simulated memory hierarchy. In the event of a memory management exception, e.g., page fault, protection exception, the MMU signals the processor which traps to the appropriate OS exception handler. The MMU also maintains and updates a page's reference and dirty bits by issuing single-byte stores to the simulated memory hierarchy when a page whose reference or change bit is not set is first referenced or written.

The PowerPC architecture includes many cache management instructions, e.g. data cache block invalidate, data cache block zero, etc., which are used in both system and user-level code. Implementing each of these instructions required significant changes to the SimpleMP coherence protocol.

We also augment the SimpleMP memory system to support coherent I/O. Both

SimpleMP [92] and SimpleScalar [14] perform I/O magically by proxying system calls and instantaneously updating a cache's contents to reflect the new memory contents. Obviously, this mechanism does not accurately model how I/O is performed in real systems. To accurately model coherent I/O, we added support to SimOS and SimpleMP for I/O controllers to initiate DMA transfers into memory and invalidate the corresponding blocks in each processor's caches.

6.2.2 Simulation Parameters for Performance Studies

We simulate 4-processor and 8-processor shared-memory, snoop-based, multiprocessor systems in our performance results. The coherence protocol is modeled after the Gigaplane-XB [22], and inclusive, write-back L0, L1, and L2 cache hierarchies are simulated. An MSI protocol is used in the L0/L1 caches, MOESI in the L2 caches, and OSI at the coherence point/L2 snoop tags, i.e. the DTAGs [22]. Contention is modeled at all levels in the memory system as well as finite microprocessor core resources. All user, library, and kernel code is directly executed by the performance model, including I/O via a coherent DMA agent. Application benchmarks are presented for 4-processor systems; to provide an additional data point, micro-benchmarks are presented for 8-processor systems. The microprocessor core is similar to SimpleMP/SimpleScalar 3.0 [92] [14] with an additional translation stage added for instruction cracking of complex PowerPC instructions. Microprocessor core resources are configured identically for all simulations; L2 cache sizes for 8-processor systems are smaller than 4-processor systems due to simulation host machine memory constraints; address network bandwidth is also scaled for different processor counts. L2-caches are pre-loaded from a system checkpoint on simulator startup for most workload configurations to mitigate cold start effects; TLBs, L0, and L1 caches are

Table 6-1: Simulated Machine Parameters. Functional unit latencies are shown in parenthesis. L0-Cache latencies indicate single cycle address generation plus a cycle for data delivery (1+1), leading to a 1-cycle load-to-use penalty. L1 and L2 cache latencies are additive (L0-miss, L1-hit latency is 1+1+4=6 cycles, L0-miss, L1-miss, L2-hit latency is 1+1+4+12=18 cycles).

Attribute	Value
Fetch/Xlate/Decode/Issue/Commit Width	8/8/8/8/8
Pipeline Depth	6 stages
BTB/Branch-Predictor/RAS	8K sets, 4-way/8K combining (bimodal and GShare)/32 entry
RUU/LSQ	256 entry/128 entry (micro-ops)
Integer	ALUS: 8 simple (1), 2 mul/div (3/12); Memory: 4 LD/ST
Floating Point	ALUS: 3 add/sub (4/4), 3 mul/div/fmac (4/4/4)
L0-Caches	I$: 64KB, 1-way, 64B lines (1+1); D$: 64KB, 1-way, 64B lines (1+1);
L1-Caches	I$: 512KB, 8-way, 64B lines (4); D$: 512KB, 8-way, 64B lines (4)
L2-Cache	Separate request/response queues for L1-I/L1-D caches; 64B-wide L1 interface, 1 cycle occupancy/txn per queue; Unified: 16MB, 8-way, 64B lines (12) (4-processor); Unified: 8MB, 16-way, 64B lines (12) (8-processor)
Memory/Cache-to-Cache	Minimum latency: 400 cycles; 50 cycles occupancy/txn, crossbar
Address Network	Minimum latency: 70 cycles, 20 cycles occupancy/txn, bus (4-processor); 10 cycles occupany/txn, bus (8-processor)
TLB	Hardware page table walker, 1-level, 2K sets, 2-way, 4Kpages
Memory Model	Sequential Consistency, similar to MIPS R10K [110], [40]
SLE	In-core buffering (RUU/LSQ hold speculative state, speculative critical section size is maximally 0.5 * RUU/LSQ size)
MESTI	Full cache-line stale storage with instant temporal silence detection; L2-cache tags used for useful validate prediction

empty at the start of each run. Precise resource configurations are given in Table 6-1. Note that we assume perfect temporal silence detection for MESTI which implies stale storage for all cache lines present in the L2 and the L2 cache tags are used for useful temporal silence prediction. We have shown throughout Chapter 6 that efficient stale storage mechanisms and L2-cache tags can be utilized as a very low-cost way to achieve essentially equivalent performance to this configuration.

6.3 Simulator Verification and Simulation Methodology

Implementing a full-system, fully-integrated timing/function simulator such as PHARMsim is an incredibly complex task. Furthermore, verification of both functional correctness and coherent/consistent memory order additionally complicates baseline simulator design as well as modeling of performance enhancement techniques; however, the correctness requirement helps keep us honest as researchers. All subtle timing races and implementation aspects must be considered, and properly handled, in order for the simulator to run an application correctly and therefore generate meaningful performance results.

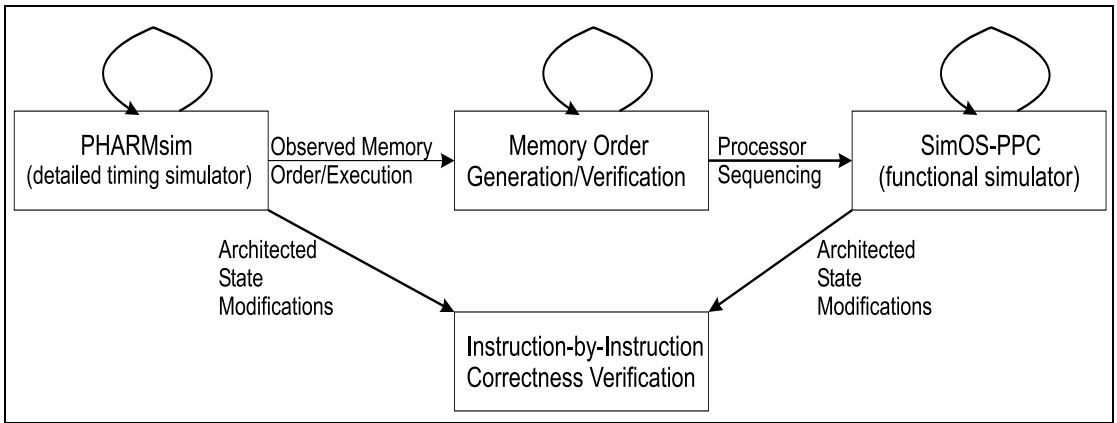

FIGURE 6-1. Block Diagram of the PHARMsim Verification Environment. Memory order and other execution-related events, e.g. interrupts, are communicated to an intermediary which determines instruction execution sequencing according to consistency model constraints. Processor sequencing information is passed to the SimOS-PPC functional simulator, with correctness verified with instruction-by-instruction checking of modifications to architected state. Execution semantics are not passed from PHARMsim to SimOS-PPC.

A complete description of simulator verification and design of mechanisms enabling verification in PHARMsim is beyond the scope of this thesis. At a high level, we enable instruction by instruction checking of modifications to architected state by mirroring execution in PHARMsim with a semantically unmodified SimOS-PPC simulator. Any mismatches in execution are indicated as errors and therefore can be traced and corrected. A block diagram of the verification environment is shown in Figure 6-1.

Observed memory order, memory consistency model, and other appropriate execution information is passed to an intermediary which verifies the consistency model has been observed. The intermediary provides instruction sequencing information to the SimOS-PPC functional simulator, and functional correctness is verified with instruction-by-instruction checking. Execution semantic information, e.g. observed load values, is not communicated between PHARMsim and SimOS-PPC; observed events in PHARMsim only indicate, through the intermediary, in what order processors in the SimOS-PPC model should run instructions. Therefore, since PHARMsim executions are verified with a semantically unmodified SimOS-PPC functional model, we are assured that PHARMsim executions follow correct PowerPC semantics. Random perturbations in event queues, memory system queues, and core/memory execution between processors on each simulated machine cycle assures fair access to shared resources and exercises simulator robustness. All ordering requirements for transactions in multiple queues between the processor and memory interconnect are always maintained.

6.3.2 Operating/Compilation Environment

PHARMsim inherits its operating environment from SimOS-PPC [56]. Therefore, it utilizes a slightly modified AIX v4.3.1 kernel which is part of the SimOS-PPC distribution as well as appropriate library code for scientific and commercial workloads. Unmodified libraries and other supporting code required for commercial workloads have been installed and can be run unmodified, provided the code is supported by AIX v4.3.1. Since PHARMsim/SimOS-PPC is a full-system simulator, it can run binaries compiled natively on 32-bit PowerPC hardware with AIX v4.3.1. We have successfully run binaries compiled with both gcc v2.95.2 as well as IBM VisualAge C/C++ v5. All scientific workloads

are compiled with gcc -04 and appropriate locking/barrier macros for AIX/PowerPC. Additional information regarding workload setup and environment can be found in Table 3-4 on page 72.

6.3.3 Simulation Methodology

Recent work has shown that performance evaluation of multithreaded programs is a complex task due to interaction between the architecture, thread scheduling within the application, and interplay with the operating system. Subtle interactions between the hardware and application program can dramatically impact simulation results. A simple example is differences in lock acquisition order brought about by slight changes in execution timing; in a certain simulation processor 1 may acquire the lock, in another simulation processor 2 may acquire the lock, causing processor 1 to spin. The operating system can make a different scheduling decision based on which processors are holding locks and an entirely different set of threads may be scheduled. A detailed discussion of these issues can be found in work by Alameldeen et al. [7] and also in our own work [67].

This complication has been called *spatial variability* [7] or, more generically, the *non-determinism problem* [67].[1] We use two simulation techniques in this thesis to handle spatial variability. The first method we use, advocated by Alameldeen et al. [7], performs multiple simulations from the same starting checkpoint with random perturbations and measures execution time for a fixed amount of work; either end-to-end application runs or fixed transaction counts. We refer to this as *statistical simulation*. The other method is a recently proposed and evaluated technique which systematically eliminates spatial vari-

1. We ignore a discussion of *temporal variability* [7] for the sake of brevity and because the issues of spatial and temporal variability are mostly orthogonal. It will become clear shortly why we focus only on spatial variability.

ability through deterministic re-execution and quantifies the sacrifice in simulator precision caused by enforcing the same execution. We refer to this method as *deterministic simulation* [67].

Implementing a deterministic multiprocessor simulator is non-trivial and therefore not explained in detail here. Conceptually, deterministic simulation records a single simulation (the *control*) and then all subsequent architectural evaluations (the *experiments*) are compared against this single execution. Since the committed instruction stream across all processors in deterministic simulation is identical to the control, we can rely on conventional uniprocessor performance metrics, such as IPC, to measure performance. Of course, we must perturb the system (sacrifice some precision) in order to assure it recreates the control execution (to buy back accuracy). We achieve this by appropriately delaying selected memory operations or instructions at interrupt arrival boundaries to ensure that the same shared-memory order and interrupt behavior is maintained. We measure how much delay was artificially inserted to ensure deterministic execution, and report this sacrifice in precision as *determinism-delay.* Advantages of this approach include achieving comparability with only a single simulation at each data point, reducing simulation cycles, immunity to cold-start and end-effects [7], and the ability to use conventional performance metrics. However, enforcing the control execution can sacrifice much precision under some scenarios [67] and should be thoughtfully considered in environments where either relaxed execution semantics or optimizations expected to significantly impact the simulated instruction stream are performed. We refer interested readers to other work [67] for a detailed discussion of advantages and caveats of deterministic simulation as compared to statistical simulation and the details of our implementation and metrics.

In the studies shown in subsequent sections for application benchmarks, we indicate statistical simulations by labeling the benchmark with an "S" on the performance graph, "D" for deterministic simulations. Baseline IPCs for application benchmarks and other program characteristics are shown in Table 6-2.

Table 6-2: Basic Application Benchmark Characteristics. Instructions are measured excluding the operating system idle loop. Temporally silent stores are measured with respect to the previous version of the cache line, i.e. those which can be exploited with MESTI. IPC cumulative across all processors.

Program	Instr.	Micro-Ops	Loads	Stores	US Stores	TS Stores	IPC
barnes-4p	1.80B	2.29B	506M	608M	117M	714K	6.73
ocean-4p	859M	984M	250M	75M	9.1M	1.67M	5.31
radiosity-4p	2.39B	3.26B	947M	647M	137M	2.84M	7.00
raytrace-4p	418M	567M	124M	94M	18M	1.04M	3.61
specweb-4p	3.0B	4.63B	1.24B	1.17B	270M	13.6M	3.86
specjbb-4p	2.0B	2.73B	600M	506M	94M	20.7M	2.41
tpc-h-4p	1.61B	3.18B	1.02B	842M	220M	38.2M	1.39

6.4 Application Benchmarks: Update Silence

As described in Chapter 4, two principle benefits can be achieved by exploiting update silence in multiprocessors: reduced communication, and associated execution latency, by eliminating USS misses and reduced address traffic due to fewer upgrade requests. As further discussed in Section 5.7.4, we use the MESTI protocol to handle pure update silent store misses under two different configurations: one where only update silence of the miss is considered and not criticality, i.e. a validate is broadcast for all update silent store misses which cannot be filtered using the simple snoop-aware validate policy; and another where the address-based prediction mechanism developed for temporal silence (Section 5.7.3) is applied to predict critical and anti-critical update silent store misses.

Performance results for 4-processor benchmarks are shown in Figure 6-2. Note that *ocean* is present twice in the results; simulated both deterministically and non-deter-

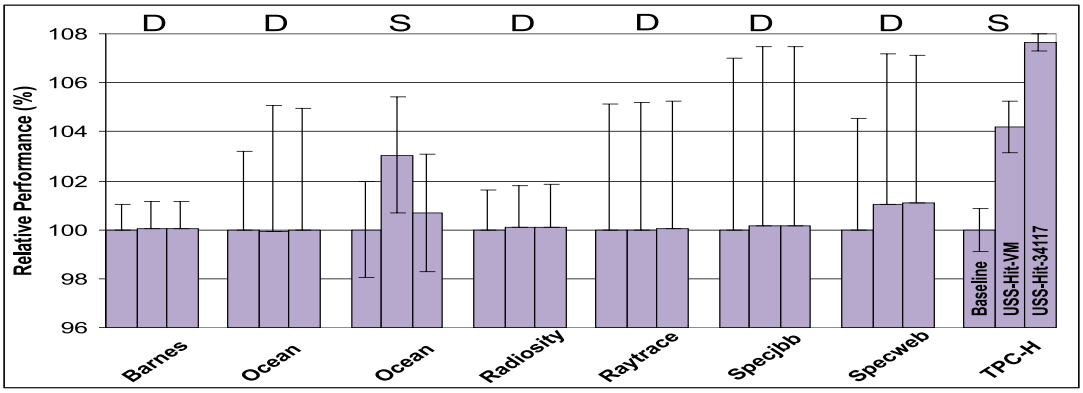

FIGURE 6-2. Performance of Exploiting Update Silent Stores in Multiprocessors. (4-processor). Performance improvement normalized to the baseline case for USS-Hit with validate on update silent store misses (Section 5.7.4, USS-HIt-VM) and including a useful validate predictor (Section 5.7.3, USS-Hit-34117) are indicated. Determinism-delay and 90% confidence intervals for benchmarks simulated deterministically or statistically is also shown.

ministically. We will return to this point shortly. Examining the performance results, we observe measurable speedups in *ocean* and *specweb*, and substantial speedup in *tpc-h*. These results correlate well with characterization data presented in Section 4.2 on miss-rate reductions possible with USS, as only *ocean* and *specweb* showed significant miss rate reductions, coupled with a relatively high cache miss rate. *Tpc-h* showed little potential in those studies; changing processor models has made this benchmark more sensitive to USS sharing patterns. Note also that no performance degradation is observed in any benchmark, indicating that both the USS-Hit-VM and USS-Hit-34117 policies are effective at minimizing performance impact due to anti-critical update silent store misses (Section 4.4).

Most performance improvement comes from eliminating communication misses between processors in all benchmarks; additional gains accrue from reduced shared address interconnect contention due to eliminated useless invalidate broadcasts (those which do not hit in any remote cache, Section 4.3). Additional benefit is achieved by eliminating transactions within the local processor between the L0/L1 data caches and the L2

for both writebacks and coherence permission changes.[2] Note that since on-chip bandwidth is plentiful in our machine model, performance improvement from these effects comes substantially from improved latency and not reduced bandwidth requirements; in bandwidth-constrained designs we expect performance may be additionally improved. Writeback reductions are similar to those presented in Section 4.2.

We include performance results for *ocean* simulated deterministically and statistically; this simple case study indicates the pros and cons of deterministic simulation. Focusing on the USS-Hit-VM results, in the deterministic simulation we observe negligible performance improvement exploiting USS; however determinism-stall increases substantially (approximately 2%). Simulating statistically indicates most of the extra determinism-delay induced (called *intrinsic determinism-delay* [67]) is speedup lost by artificially enforcing the same control and experiment executions. Statistical simulation relaxes this requirement, at the expense of additional simulation bandwidth, and additional speedup is obtained. We assume a similar conclusion holds for *specweb* where intrinsic determinism-delay increases by approximately 3%; however, we cannot verify this assertion because our configuration of this workload does not run a fixed number of transactions.

Figure 6-3 indicates observed address transactions for each configuration. The benchmarks indicating substantial potential for USS to eliminate communication, coupled with substantial transaction rates, i.e. *ocean*, *specweb*, and *tpc-h*, are also those where the greatest performance improvement is achieved. The behavior of *ocean* simulated statistically is noteworthy; we observe that the critical update silence predictor is eliminating the

2. As discussed in Section 6.2.2, the L0-I, L1-I, L0-D, and L1-D caches only implement the MSI protocol, with full MOESI implemented at the L2.

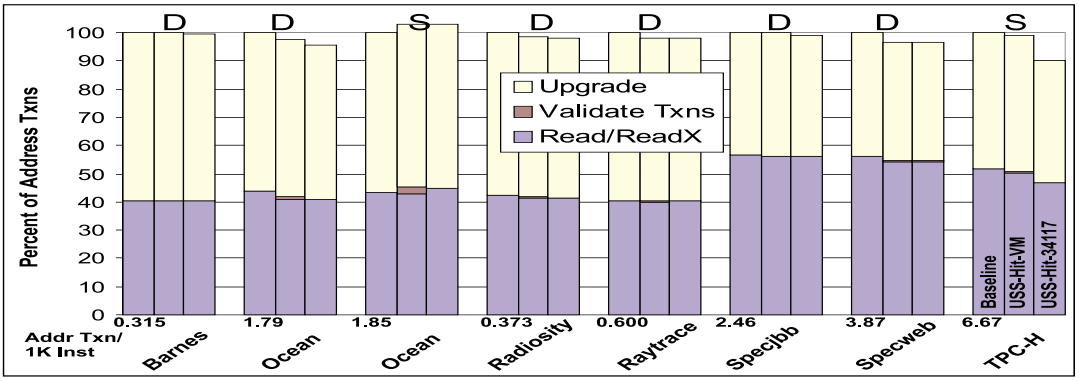

FIGURE 6-3. Address Transactions Observed Exploiting Update Silent Stores. (4-processor) Results normalized to baseline case for USS-Hit with validate on update silent store misses (Section 5.7.4, USS-HIt-VM) and including a useful validate predictor (Section 5.7.3, USS-Hit-34117) are indicated. Address transaction rate per committed instruction is also shown.

vast majority of validates, sacrificing substantial speedup opportunity indicated for the USS-Hit-VM configuration in Figure 6-2. Characterization data in Section 4.4 indicated that most update silent store misses in *ocean* are indeed critical, thus conservatism of the predictor is hampering performance potential.

6.5 Understanding Performance Potential Through Microbenchmarking

Microbenchmarks are a useful way to provide intuition into performance results of more complicated programs. Since we use the same mechanisms, i.e. the MESTI protocol, to exploit both update silence and temporal silence (see Section 5.7.4) we only present a microbenchmark targeted toward temporal silence. Furthermore, since speculative lock elision (SLE) is an important piece of related work to temporal silence, we explore the performance potential of MESTI, and compare it to SLE, with this microbenchmark; application benchmarks will be presented in subsequent sections. Of course, a microbenchmark or even a suite of microbenchmarks, cannot capture all interesting interactions between machines and workloads but the example we present provides valuable intuition for comparing the two approaches. We focus on the idiom of critical sections and spin-locks as that is the idiom targeted by SLE; it is also an intuitively obvious candidate

for exhibiting temporal silence in many programs.

As discussed in Section 5.4.3, SLE can achieve performance benefit from two major sources: additional exposed concurrency by removing artificial dependence on the lock variable (enabling concurrent execution of non-conflicting critical sections) and eliminating misses on the lock variable itself (in the case of non-concurrent execution of critical sections). MESTI, on the other hand, cannot remove artificial dependence on the lock variable; thus it can only achieve benefit from eliminating misses on the lock variable itself. However, MESTI can achieve additional benefit not possible with SLE; namely, temporal silence occurring in contexts outside the idiom SLE attempts to exploit. To our knowledge, SLE has only been targeted toward synchronization primitives. However, empirical data from commercial benchmarks, e.g. Section 5.10.1 *specjbb* principally—exhibited to a lesser extent by other benchmarks, indicates temporal silence occurs outside synchronization primitives.

Therefore, we measure performance sensitivity of a base machine, one implementing SLE, and another implementing MESTI, to three parameters:

χ: Probability of conflicting accesses performed within a single critical section.

θ: Probability of temporal silence through idioms other than synchronization.

δ: Degree of parallelism inherent in the benchmark itself.

χ is a program attribute, describing the probability of conflicting data accesses within the same critical section, intended to indicate successful partitioning of the parallelized problem. θ is also a program attribute, indicating the degree to which temporal silence occurs outside idioms exploitable by SLE.

δ is a more complicated parameter, as it encapsulates both program attributes and

machine attributes. The program attribute portion of delta is characterized by the portion of the program which is inherently serial (α), a pure parallel portion with no potential for data conflicts (β), and the degree of explicit parallelization of the algorithm itself (ε). Examples of α are serial initialization phases or mutually-exclusive region accesses, i.e. critical sections; examples of β are purely thread local computations, e.g. local data structure manipulation; examples of ε include locking granularity. The machine attribute of δ, which we call π, indicates the ability of the underlying hardware to exploit the program attributes α, β, and ε and translate them into improved program performance, i.e. the number of parallel processing elements. In order to clarify the meaning of each parameter, we present partial pseudo-code for our microbenchmark in Figure 6-4 with each component labeled.

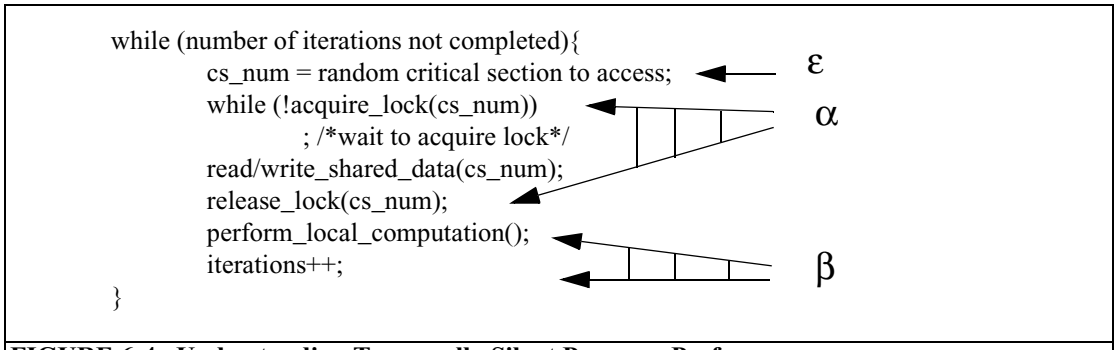

FIGURE 6-4. Understanding Temporally Silent Program Performance.

Our microbenchmark consists of each thread performing a fixed number of accesses to multiple critical sections, each critical section protecting a set of shared counters. The action within the critical section leads to a single increment of a shared counter (similar to microbenchmarks presented in Rajwar's thesis [90] illustrating the performance potential of SLE). The application code part of the critical section is part of serial execution (α) and local computation is completely parallel (β). The number of critical sections which can be accessed on each iteration indicates the degree of explicit paral-

lelism inherent in the benchmark (ε).

The machine attribute π indicates the number of processors performing the main computation loop. It should be intuitively obvious that parallel execution of this application is determined by the interplay of all sub-parameters of δ; the π portion simply determines how many critical sections we can attempt to run in parallel at a given time. Hence, varying any sub-parameter of δ can effectively indicate performance scaling in any of the other parameters. To keep the number of simulations tractable, we fix α, β, and π and only vary ε as an effective and simple way to indicate performance sensitivity to δ. Practically speaking, this means the size of each critical section is fixed, the time outside each critical section is fixed, and the number of processors is fixed, with the only variable being the number of critical sections protecting the shared counters. With this understanding, we present complete pseudo-code for our microbenchmark in Figure 6-5. The meaning of each parameter and additional function added over Figure 6-4 is indicated in the caption. Note that parameters indicating the likelihood of conflicting data accesses in the critical section (χ) and the occurrence of temporal silence (θ) should be considered probabilities on each iteration. Note that when θ is non-zero, the probability of data conflicts to shared counters are reduced by θ so only a single conflict (to either shared counters or temporally silent data, but not both) occurs on each iteration so the intended meaning of χ is preserved.

Performance results for this microbenchmark for varying χ, ε, and θ are shown in Figure 6-6 for varying values of χ in each graph with θ=0% and Figure 6-7 for varying values of θ in each graph with χ=θ. Each thread runs 20,000 iterations of the microbenchmark code; 160,000 iterations total across processors.

```
/*initialize shared data structures, fork π threads*/
while (number of iterations not completed){
        cs_num = random() % ε;                                  ε
        while (!acquire_lock(cs_num))                           α
                ; /*wait to acquire lock*/
        read/write_temporally_silent_data(cs_num, θ);
        counter_conflict = (χ−θ);
        read/write_counter_data(cs_num, counter_conflict);
        release_lock(cs_num);
        random_nonuniform_backoff();                            β
        iterations++;
}
/*join all threads*/                                            α
master_thread_compute_all_counters_total();
```

FIGURE 6-5. Microbenchmark Pseudo-Code. A read/write data conflict of temporally silent data occurs with probability θ (random, uniformly distributed), a data conflict within the critical section to shared counters occurs with probability (χ-θ) (χ random, uniformly distributed). Random, non-uniform backoff is used outside the critical section to ensure fairness (as discussed in Culler and Singh [31] for simple spin-locks.) Empirical evidence (not detailed here) indicates random non-uniform backoff is only required for ε=1, but we use the same backoff parameters across all values of ε. All data structures are appropriately padded to eliminate conflicting accesses to the lock variable, temporally silent data, and shared counters due to coherence granularity. Dynamic critical section size is small enough to fit entirely within in-core buffering for SLE.

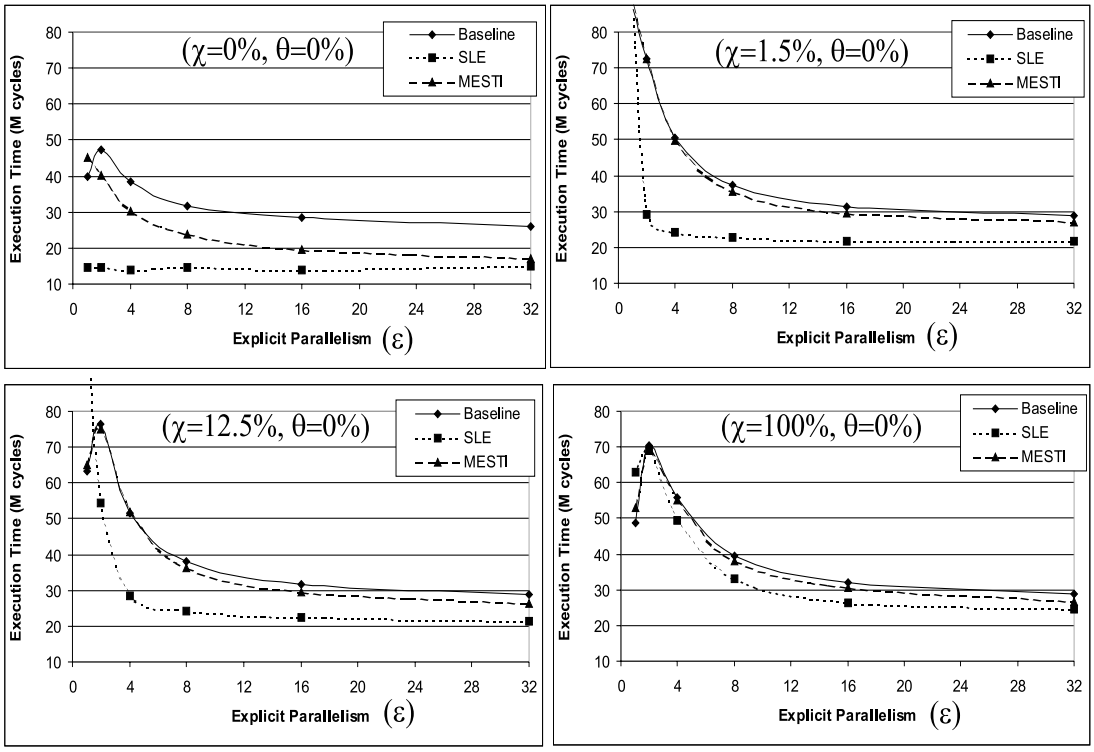

FIGURE 6-6. Microbenchmark Execution Time for Varying Values of χ and ε.

Examining Figure 6-6, for $\chi = 0$, we observe that SLE achieves near constant performance irrespective of ε. This is expected; in the absence of data conflicts SLE enables all processors to execute critical sections concurrently, regardless of explicit parallelism specified by the programmer. MESTI performs similarly to the base case for small ε. This occurs because MESTI cannot allow concurrent execution of critical sections; all intermediate value and temporally silent stores must be visible. However, we observe that as ε is increased, MESTI continually improves its performance over the baseline, approaching the performance of SLE. Asymptotically, we expect MESTI and SLE to deliver the same performance[3] for increasing ε because, in the absence of races for the same critical section, both schemes can successfully eliminate the observed latency for the lock acquire. This indicates that, for well-tuned applications without data conflicts–with relatively uncontended critical sections–SLE and MESTI can deliver the same performance as long as the temporally silent pairs captured by both schemes are identical.

Examining Figure 6-6 for $\chi = 1.5\%$, a low but finite conflict probability, similar trends are observed but are less pronounced. An interesting observation is the reduced performance increase delivered by SLE as compared to the baseline case; with this low conflict probability, the loss in performance is not due to SLE restarts. Rather, because data within the critical section is actually shared, even though SLE can speculate through nearly all critical sections concurrently, actual data sharing slows execution due to cache-to-cache transfers of data within the critical section. MESTI suffers from the same limitation, but again, asymptotically performance approaches SLE.

---

3. This assumes the impact of additional address traffic in MESTI due to upgrade/validate pairs does not materially impact performance, i.e. a non-bandwidth constrained environment, only latency constrained.

Similar trends are observed for larger χ (12.5% and 100%), with SLE performance consistently degrading toward baseline performance. For $\varepsilon = 1$ and $\varepsilon = 2$, we observe degradation over baseline performance due to excessive SLE restarts using a restart threshold of 1, as done in Rajwar's thesis [90].

Note that for $\varepsilon = 1$, baseline performance is actually better than SLE and MESTI under some scenarios, which may be unexpected. This occurs because of interaction between the microbenchmark and the coherence protocol with relatively small β. As explained in Culler and Singh [31], spin locks can be unfair when β is small, as compared to the speed of the address network, because a processor with a valid copy, i.e. the previous lock holder, of the data can repeatedly acquire the single lock successfully, disallowing all other processors access to the critical section. When this occurs, all data protected within the critical section tends to become cache resident on the processor repeatedly acquiring the lock, allowing it to perform many iterations quickly, as compared to the fair case where data transfers within the critical section are more likely to be cache-to-cache. Non-uniform random backoff improves fairness, at the expense of reducing sensitivity to communication latency, i.e. through increase of β; since we are interested in accentuating communication performance, we choose a relatively small backoff value. SLE and MESTI tend not to exhibit the same unfairness because of different interactions with the coherence protocol.

In the previous discussion, we never observed a case in which MESTI can actually outperform SLE. Indeed, if all temporal silence occurred via idioms detectable with SLE, provided sufficient machine buffering and the absence of data conflicts, we always expect SLE to outperform MESTI because it avoids making intermediate values and temporally

silent values visible through the coherence protocol. However, proper idiom detection may be difficult for some programs, as we will describe in Section 6.6.5; furthermore, empirical evidence indicates significant temporal silence can occur outside of memory locations touched with architected synchronization primitives (Section 5.10.1). Therefore, we present performance sensitivity of SLE and MESTI with varying degree of temporal silence occurring within the critical section which cannot be detected or exploited via SLE. The results are shown in Figure 6-7.

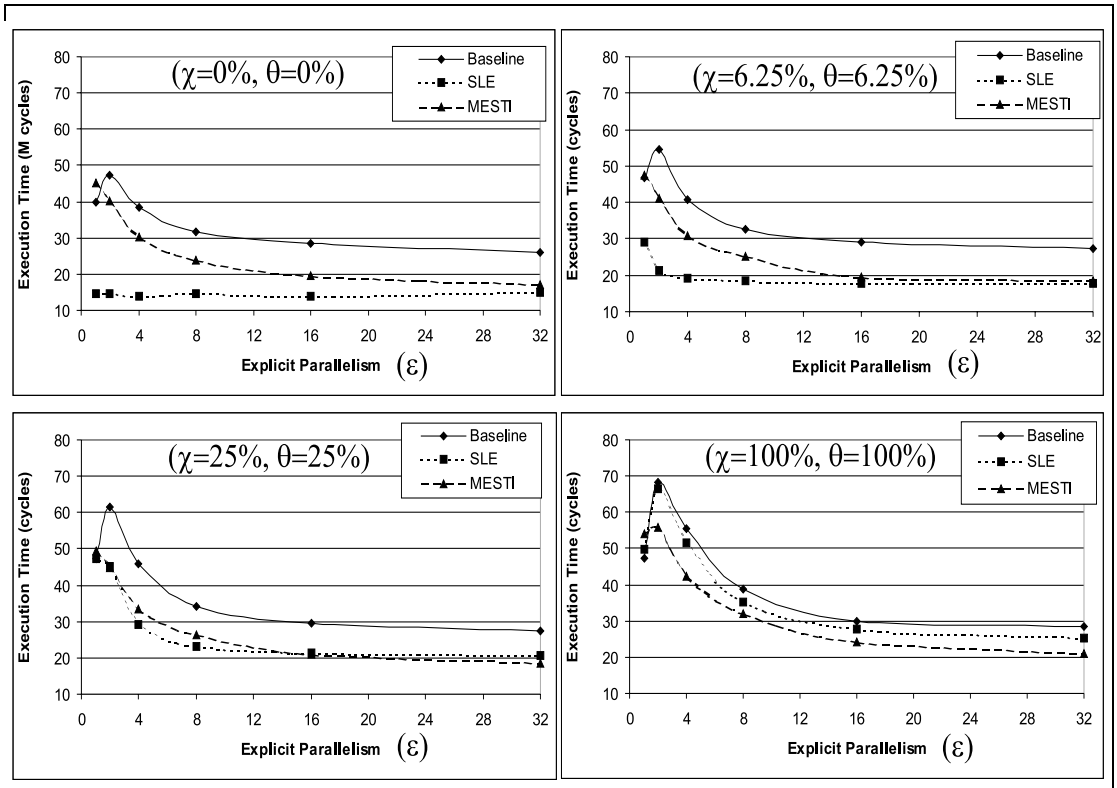

FIGURE 6-7. Microbenchmark Execution Time for Varying Values of θ and ε.

Similar trends (as a function of ε) are observed as in Figure 6-6. Note that SLE performance in all cases is similar to Figure 6-6 for equivalent χ values; with SLE the temporally silent data conflicts are not detected and thus behave similarly to shared counter updates. As temporally silent data conflicts (θ) increase, MESTI can exploit both temporal silence of the lock variable as well as data within the critical section. Its perfor-

mance continues improving compared to SLE, essentially equalling or surpassing it for ε = 16 and ε = 32 for θ = 25%, and surpassing it substantially at all data points for θ = 100%. Note that a similar performance differential can develop between SLE and MESTI, not only due to temporal silence outside SLE's idiom as indicated with θ, but also in cases where temporal silence occurs beyond the reach of speculation due to insufficient buffering, barriers to speculation, or temporal silence occurring across multiple critical sections. As shown in Section 5.5.3, temporally silent pair distance can be substantial, especially in commercial workloads, implying greater difficulty in collapsing temporally silent pairs atomically. Also, as discussed in Section 5.4.3, longer critical sections have higher data conflict probability due to actual conflicts and also perceived conflicts through the coherence protocol due either to false sharing or exclusive prefetching schemes. As critical section length increases, conflict probability increases, lost opportunity due to speculative execution increases, and restart penalty may also increase, tending to degrade SLE performance. MESTI provides better algorithmic stability in the face of such issues. We will return to this discussion when examining application benchmarks.

6.6 Application Benchmarks: Temporal Silence

As illustrated in Section 5.2, exploiting temporal silence with the MESTI protocol provides an additional avenue for eliminating remote communication misses. We now present performance data for our implementation of the MESTI protocol in PHARMsim and compare its performance with load value prediction as well as SLE.

6.6.1 Basic MESTI Implementation

We present performance data for the basic MESTI protocol with Snoop-Aware Validate policy (Section 5.7) in Figure 6-8. Recall that this policy broadcasts a validate at

each occurrence of temporal silence to cache lines which were ever shared between processors in the system.

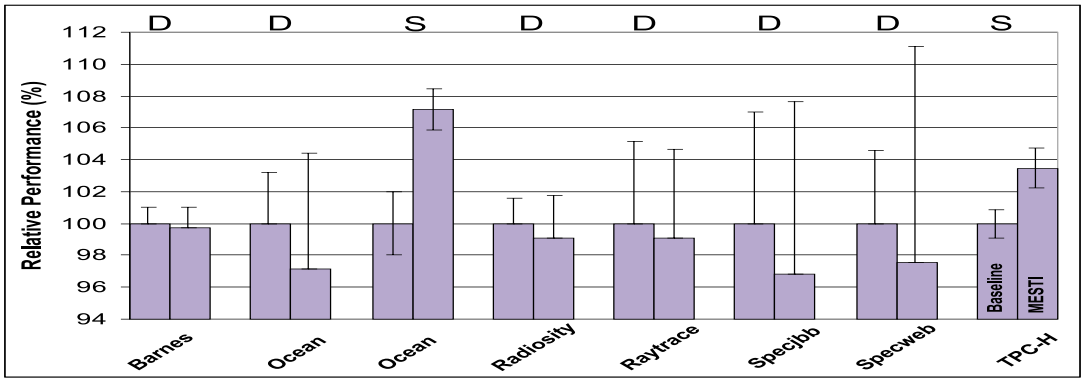

FIGURE 6-8. Performance of MESTI Protocol Exploiting Temporal Silence. (4-processor) Performance improvement normalized to the baseline case for the basic MESTI protocol exploit update silence and temporal silence is indicated. Determinism-delay and standard deviation for benchmarks simulated deterministically/statistically is also shown.

We see significant speedups in statistically simulated *ocean* and *tpc-h*. Slowdowns are reported in all other workloads, with those measured for deterministically simulated *ocean*, *specjbb*, and *specweb* being significant. In *ocean*, it is reasonable to conclude that most of the reported slowdown is actually due to intrinsic determinism-delay; a similar conclusion is also likely, but as explained in Section 6.4, unverifiable, for *specweb*. Because determinism-delay in *specjbb* for both simulations is substantial, the result is largely inconclusive.

As discussed in Section 5.7, a potential performance issue with the basic MESTI protocol is the large percentage of anti-critical temporally silent stores. Since many temporally silent writes are not last writes, many validate/upgrade pairs may occur without preventing any remote sharing misses. This can degrade performance due to both latency induced by coherence transitions on the local processor and also additional coherence traffic in the system. We indicate the magnitude of this effect in our detailed simulation envi-

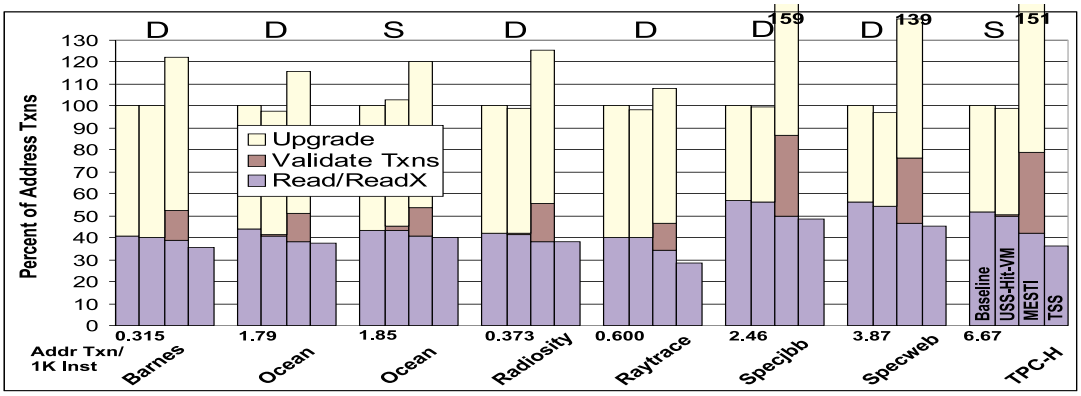

FIGURE 6-9. Address Transactions Observed with MESTI Protocol . (4-processor)
Results normalized to baseline case for USS-Hit-VM (Section 6.4), the basic MESTI protocol, and the ideal TSS case are indicated. Address transaction rate per committed instruction is also shown. Note that validate and upgrade transactions cannot be presented for TSS since it is only a hypothetical result indicating maximal data transfer reduction.

ronment in Figure 6-9.

Mirroring results presented in Section 5.7, we observe that the simple Snoop-Aware Validate policy increases address transactions over the baseline case significantly, ranging from 5% to 59%. Most noteworthy is the contribution of useless validates, which is determined by comparing the height of the Read/ReadX plus Validate components for MESTI to the Read/ReadX component of the baseline.[4] For all benchmarks, we observe a substantial contribution to address transactions from useless validates. Furthermore, the substantial increase in upgrade transactions over the baseline case further indicates many of these useless validates are also anti-critical. However, in many cases (see Figure 6-8), the reduction in communication misses offsets these effects. In our machine model, address bandwidth is relatively plentiful, therefore most performance degradation comes from additional memory latency observed by the processor core for additional coherence state transitions; in more bandwidth-constrained environments the negative performance

4. This comparison does not precisely determine the number of useless validates because a single validate can prevent multiple communication misses, as discussed in Section 5.7. However, it serves as a useful first-order approximation. We call a validate *useless* if broadcasting it does not prevent a remote miss.

impact will likely be more substantial.

As an orthogonal issue, we can use this data to provide an indication of whether our initial studies conducted in Chapter 5 under a different machine model reliably indicated the behavior of MESTI in the detailed simulation environment. Examining the data traffic reductions in Figure 6-9 (the Read/ReadX component), we observe similar reductions in data traffic as those presented in Figure 5-1 on page 109 and Figure 5-7 on page 120 for TSS and MESTI, respectively. This indicates other studies performed in Chapter 5 are likely applicable to modern, out-of-order, processor cores as well.

6.6.2 Enhanced MESTI Implementation

As discussed in Section 5.7 and briefly revisited in the previous section, a substantial fraction of useless and anti-critical validates can be eliminated by using critical silence prediction. Using prediction, confidence in the usefulness of validates can be achieved, thus minimizing excess validates at the cost of sacrificing some data transfer elimination possible with the basic MESTI implementation. We discussed a detailed design for our critical silence predictor in Section 5.7.3. Initial studies in that section indicated that the 3-4-1-1-7 configuration provided a good trade-off between eliminating useless validates without sacrificing substantial opportunity, thus we only explore this configuration in detail within the performance model. Our enhanced MESTI implementation includes the 3-4-1-1-7 critical silence predictor and a 32-cycle delay queue (Section 5.7.1). We have also added an additional enhancement of the internal coherence protocol between the L1 and L2 caches; when validate broadcasts are removed by the critical silence predictor, exclusive access permission is returned to the L1 cache, thus improving store commit latency. Performance results for this enhanced MESTI protocol are shown in Figure 6-10.

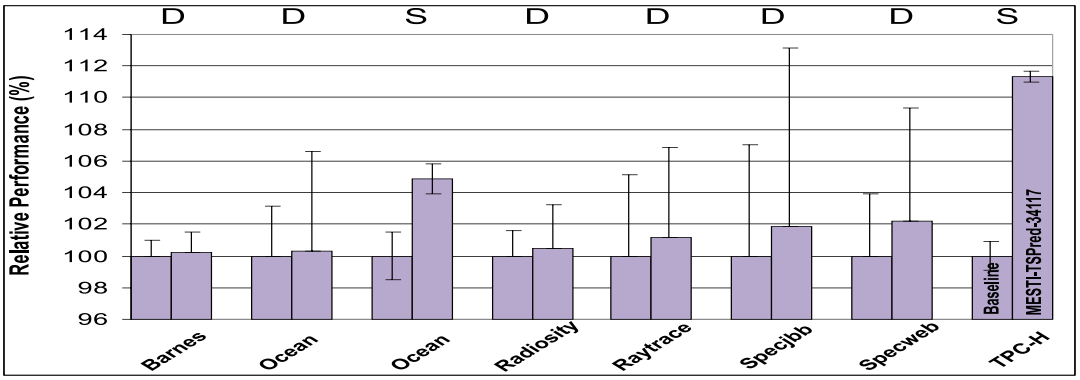

FIGURE 6-10. Performance of Enhanced MESTI Protocol. (4-processor)
Performance improvement normalized to the baseline case for an enhanced MESTI protocol exploiting critical silence through coherence prediction is indicated. Determinism-delay and 90% confidence intervals for benchmarks simulated deterministically or statistically is also shown.

Performance improvements over the basic MESTI implementation are observed for all benchmarks except *ocean* simulated statistically, indicating the benefit of eliminating useless validates outweighs the lost opportunity in communication reduction. In the case of *specweb* and *specjbb*, we observe substantial intrinsic determinism-stall, implying again that substantial performance potential may be lost due to deterministic simulation, as discussed in previous sections. However, measurable speedup is still indicated for both benchmarks. In the case of *ocean*, most lost performance opportunity occurs due to sacrificed benefit from update silent stores, as opposed to temporally silent stores, due to conservatism of the predictor; we discussed this in detail in Section 6.4. For this particular benchmark, a hybrid scheme bypassing the predictor for update silent store misses but using it for temporal silence might gain back the lost performance potential. However, we also observed in Section 6.4 that *tpc-h* benefits substantially from critical update silent store miss prediction, therefore, such a configuration may not prove beneficial in general. We do not explore such a hybrid scheme for the sake of brevity.

Figure 6-11 shows address transactions observed with critical silence prediction, compared to the baseline case and also ideal data transactions required with perfect TSS.

Comparing to the basic MESTI protocol (Figure 6-9), we observe a substantial reduction in useless validates; all benchmarks show less than a 5% increase against the baseline case. This data also shows, by examining the Read/ReadX component, the reduction in communication miss opportunity incurred with the prediction mechanism. These results closely mirror results from characterization studies in predictor design presented in Section 5.7.3. For example, focusing on the commercial workloads, we observed previously that *specweb* was least amenable to critical silence prediction because it sacrificed the most opportunity for communication reduction; the same result is indicated in Figure 6-11. Similar conclusions hold for other benchmarks, again showing that previous characterization studies are closely mirrored within the detailed simulation environment in spite of significant machine model differences.

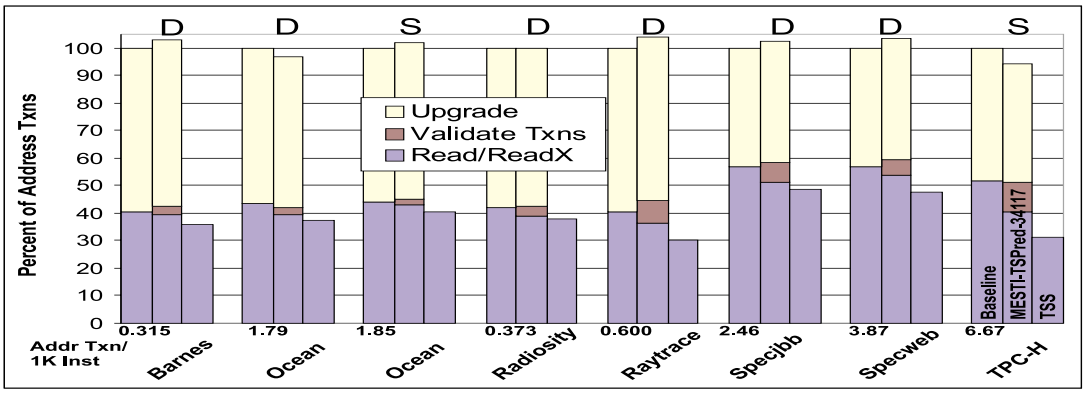

FIGURE 6-11. Address Transactions Observed with Enhanced MESTI Protocol. (4-processor) Results normalized to baseline case for MESTI-TSPred-34117 (Section 5.7.3) and the ideal TSS case are indicated. Address transaction rate per committed instruction is also shown. Note that validate and upgrade transactions cannot be presented for TSS since it is only a hypothetical result indicating maximal data transfer reduction.

6.6.3 Comparison with Load Value Prediction (LVP)

In Section 5.4.4, we discussed using load value prediction (LVP) with tag-match invalid cache lines as an avenue for exploiting TSS. This mechanism improves upon MESTI and SLE in that it can capture all TSS misses, all TSS false sharing misses, and a

subset of TSS true sharing misses, as explained in Section 5.4.4. However, a caveat of this approach is its inability to avoid data transfers for data exhibiting temporal silence; since tag-match invalid data is used as value predictions, data must be transferred in order for predictions to be verified. Therefore, we expect no data traffic reductions as compared to the baseline scheme for this approach. In contrast, MESTI and SLE can exploit many cases of TSS and also avoid data transfer. This is achieved via explicit validates in MESTI and atomic temporally silent store pair elision in SLE.

Additionally, because LVP exploits TSS at the consumer of the data, as opposed to the producer for MESTI and SLE, the latency incurred to verify the prediction is partially exposed to the consumer. In the case of successful LVP, if no additional instruction-level parallelism (ILP) or memory-level parallelism (MLP) is exposed through correct early value delivery, we expect no performance benefit to result; the machine will simply stall waiting for data transfer to verify the prediction within the speculative execution window, similar to the base case where the load is a cache miss. In the case of unsuccessful LVP, i.e. value misprediction, two performance penalties can result: additional wrong-path memory references leading to cache pollution and increased address traffic and execution penalties due to mispredicted speculative state recovery. In the base case without value prediction, these penalties are not present because the execution window will stall waiting for data to be delivered by the memory system before propagating it further in execution.

We implement machine-squash recovery for value misprediction to assure coherence and the load/store queue snooping approach described by Martin et al. [76] to assure memory consistency in machines implementing value prediction. Furthermore, the machine can exploit tag-match invalid cache lines which hit in any level of on-chip cache:

L0-D, L1-D, or L2. In the case of lower level misses, e.g. L0-D, L1-D miss, L2 hit, data transfer of invalid data is modeled over all interfaces. An additional transfer occurs to the lowest level, L0-D, through all upper levels when valid data arrives from the memory system. This effect increases on-chip memory traffic even in the case of successful LVP, but would likely be required in real implementations. Performance data for this approach is presented in Figure 6-12. Note that all scientific workloads are simulated statistically in these results, in contrast to previous sections, since substantial determinism-stall was observed for LVP. *Specweb* and *specjbb* are still simulated deterministically because they do not run for fixed transaction counts.

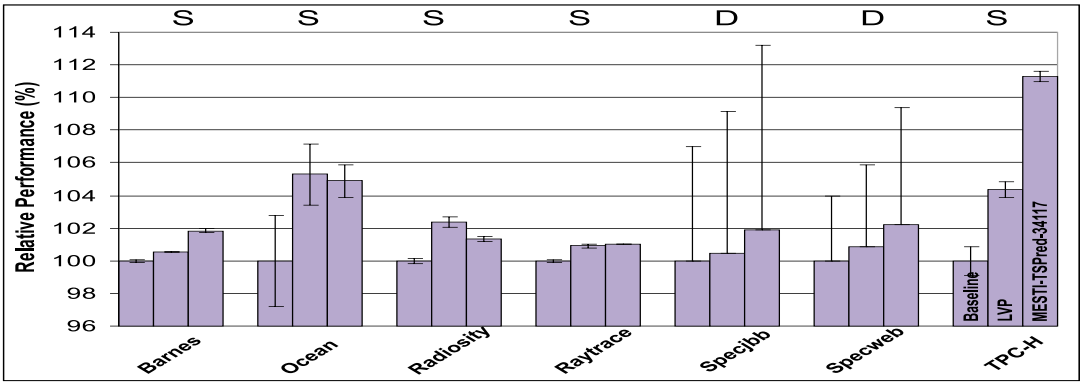

FIGURE 6-12. Performance of Load Value Prediction (LVP). (4-processor)
Performance improvement normalized to the baseline case for load value prediction with invalid cache line data (LVP) and the enhanced MESTI protocol exploiting critical silence through coherence prediction is indicated. Determinism-delay and 90% confidence intervals for benchmarks simulated deterministically or statistically is also shown.

We observe measurable performance improvements in *ocean*, *radiosity*, and *tpc-h*. In *ocean* and *radiosity*, performance slightly surpasses our enhanced MESTI implementation; however *tpc-h* shows a large performance gap between LVP and enhanced MESTI. In the other workloads, slight performance improvements are indicated, none of which exceeds the performance of enhanced MESTI. As discussed, although the candidate set of references which can be exploited by LVP is larger than that for MESTI, miss latency may

still be partially exposed. Even with an aggressive machine configuration (256 entry RUU, see Table 6-1), the long miss latency cannot be entirely hidden. Note that simply increasing the instruction window further may not improve performance; other cache misses (not to tag-match invalid lines or tag-match invalid but unsuccessful LVP), other execution-serializing events (pipeline draining conditions), or memory consistency-related squashes may also cause execution to stall before additional ILP/MLP can be exposed. Other speculation conditions, such as branch misprediction, may not cause execution to stall because they can be correctly resolved with tag-match invalid data as long as the value provided is correct.

As described, a potential performance caveat with LVP is misspeculation recovery. We use a conservative, machine-squash approach common in many microarchitectures as that is the only mechanism available in the current performance simulator. Some modern microarchitectures, e.g. Pentium4 [37], implement selective recovery mechanisms, which may reduce performance penalty due to LVP-related recovery.[5] In experiments not detailed here for brevity, we explored an unimplementable, near-perfect, LVP mechanism which virtually eliminated LVP-related squashes. Performance results of this study did not differ materially from those presented here, indicating that machine-squashing recovery was not materially degrading LVP's performance.

We now examine the impact of LVP on memory-system traffic. Figure 6-13 indicates address transactions observed with LVP as well as baseline, enhanced MESTI, and ideal TSS results for comparison. In all cases except *tpc-h* we observe an increase in both

5. Actually, the selective recovery mechanism in Pentium4 (and similar mechanisms) do not directly support value speculation as they are designed for speculative scheduling-related recovery [12] and assume fixed-latency misspeculation events, e.g. cache hit/miss speculation. Implementing general selective recovery mechanisms which support arbitrary latency misspeculation events is an orthogonal issue to this thesis.

data requests (Read/ReadX) and address-only (Upgrade) transactions for LVP as compared to the baseline. As discussed, we expect traffic for LVP to be similar to the baseline case; additional transactions are most likely caused due to additional wrong execution-path effects. The performance results indicate that these additional transactions are being successfully translated into performance within the processor core through improving ILP and MLP. The slight decrease in transactions for *tpc-h* is unexpected, but can easily be accounted for via improvement in other second-order factors relating to speculative execution since performance increases in this benchmark.

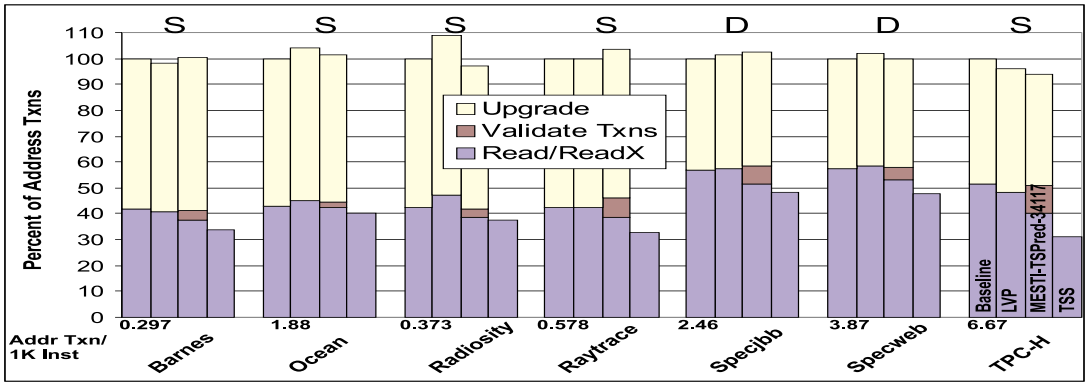

FIGURE 6-13. Address Transactions Observed with Load Value Prediction (LVP). (4-processor) Results normalized to baseline case for load value prediction with invalid cache line data (LVP), MESTI-TSPred-34117 (Section 5.7.3), and the ideal TSS case are indicated. Address transaction rate per committed instruction is also shown. Note that validate and upgrade transactions cannot be presented for TSS since it is only a hypothetical result indicating maximal data transfer reduction.

6.6.4 Combining Enhanced MESTI and Load Value Prediction (LVP)

We showed in Section 6.6.3 that while the candidate set of sharing events which LVP can capture is the greatest of all methods we discuss (TSS misses, false sharing misses, and some true sharing misses), exposing miss latency to the processor core leads to reduced performance as compared to the non-speculative MESTI protocol. In this section, we explore performance of the enhanced MESTI protocol (Section 6.6.2) combined with LVP (Section 6.6.3), referring to this as MESTI+LVP. Since the set of sharing events

captured by both methods are complimentary in many cases, we expect a performance increase when combining the two methods over each used individually. We will verify this intuition in short order.

Figure 6-14 shows the performance of MESTI+LVP, as well as baseline, LVP, and enhanced MESTI performance for comparison. Note that all scientific workloads are simulated statistically in these results, as in Section 6.6.3, since substantial determinism-stall was observed for LVP. *Specweb* and *specjbb* are still simulated deterministically because they do not run for fixed transaction counts.

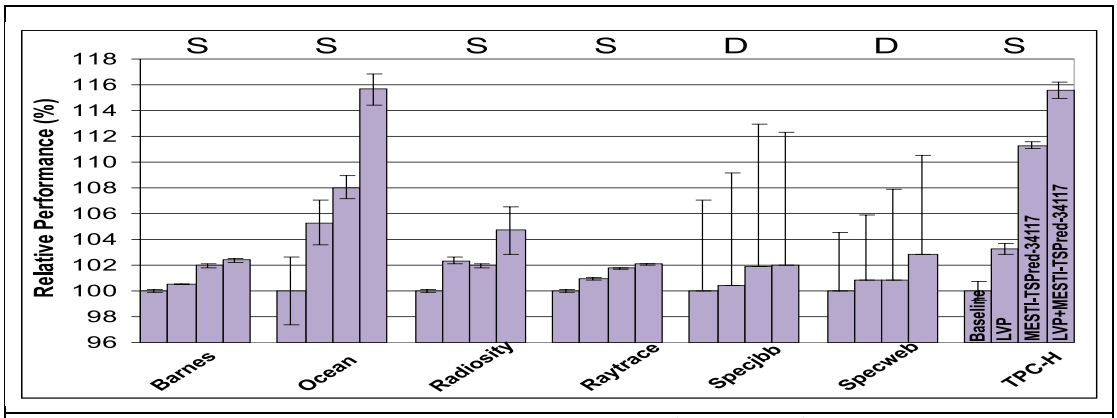

FIGURE 6-14. Performance of Enhanced MESTI + LVP. (4-processor)
Performance improvement normalized to the baseline case for load value prediction with invalid cache line data (LVP), the enhanced MESTI protocol, and both methods combined is indicated. Determinism-delay and 90% confidence intervals for benchmarks simulated deterministically or statistically is also shown.

We observe performance increases for MESTI+LVP in all cases over either LVP or enhanced MESTI in isolation, as described previously. In *ocean* and *tpc-h*, the total speedup over the baseline is substantial, 16%. In all other cases, non-negligible performance increases from 2-5% are shown.

More interestingly, in all benchmarks the performance improvement from MESTI and LVP in isolation is almost completely additive. For example, in *tpc-h*, speedup from LVP alone is 3.5% and enhanced MESTI alone is 11.5%; MESTI+LVP speedup is approx-

imately 15%. This indicates that MESTI and LVP are achieving performance benefit from a disjoint set of sharing misses, although there is substantial overlap in the candidate set of sharing misses they can capture (TSS misses). We have shown previously that MESTI eliminates most TSS misses in isolation, leading to the conclusion that LVP is not achieving substantial performance benefit from these misses. Rather, LVP is utilizing both false sharing and true sharing misses and successfully exposing additional ILP/MLP for only these references, a slightly unexpected phenomenon.

A possible explanation lies in the nature of some TSS sharing patterns. Consider the case of lock variables. For a well-tuned program locks are uncontended; however, miss latency is incurred on each lock transfer. If we assume the common test and set type lock, this latency comes from two events: the initial Read access to determine whether the lock is held and a subsequent Upgrade to perform lock acquisition. MESTI can completely eliminate the Read latency through the validate transaction, thus accelerating the lock transfer.

However, LVP cannot improve lock transfer performance, even in the case of successful LVP. As soon as tag-match invalid data is accessed, the processor returns the data to the core but also issues a Read to verify the prediction. The processor uses the tag-match invalid data to ensure branch predictions/etc. are resolved indicating the lock should be acquired; however, since the LVP has not been verified, the acquiring store never reaches the commit stage, and the Upgrade is not performed. Once LVP is verified, the load commits and then the store commits, leading to the Upgrade. However, it should be apparent that no latency associated with the lock was hidden.[6] Note further that resolving branch predictions/etc. early with LVP is likely not effective in exposing additional

ILP/MLP; in the case of uncontended locks, the branch predictor will successfully predict the lock-acquire program path.

However, since all TSS is not due to lock variables (Section 5.10.4), this explanation is insufficient to cover all scenarios. Due to the complex interactions taking place within the processor core with LVP, providing additional insight is difficult.

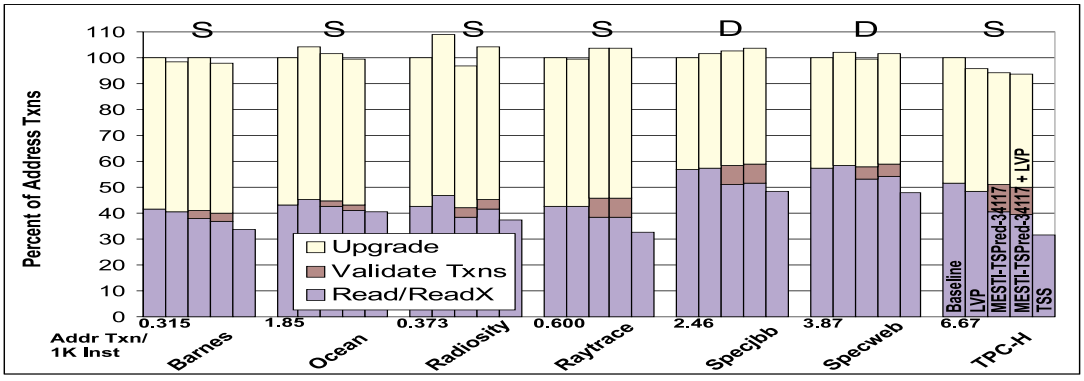

FIGURE 6-15. Address Transactions Observed with Enhanced MESTI + LVP. (4-processor) Results normalized to baseline case for LVP, MESTI-TSPred-34117 (Section 5.7.3), MESTI+LVP, and the ideal TSS case are indicated. Address transaction rate per committed instruction is also shown. Note that validate and upgrade transactions cannot be presented for TSS since it is only a hypothetical result indicating maximal data transfer reduction.

Figure 6-15 indicates address transactions observed with MESTI+LVP as well as baseline, LVP, enhanced MESTI, and ideal TSS results for comparison. No clear trend emerges from the data; since enhanced MESTI tends to decrease data transactions (i.e Read/ReadX) at the expense of additional address-only transactions and LVP tends to increase data transactions, these opposing forces lead to no clear result. In most cases, MESTI+LVP has fewer data transactions as compared to LVP with address-only transactions, i.e. Upgrade and Validate, being nearly equal to enhanced MESTI. Note that even

6. Exclusive prefetching within the core for the store will likely be ineffective to improve this; since the Read transaction is launched as soon as tag-match invalid data is accessed, the cache line will be in a pending coherence state and any other requests, e.g. an exclusive prefetch, will be NACKed until data is returned. Store write-fault predictors allowing LVP verification to also prefetch exclusive permission or elaborate coherence mechanisms allowing the Upgrade before data is returned might be useful; we do not explore such mechanisms for brevity.

with both methods combined, address traffic does not increase more than a 2-3% over the base case.

6.6.5 Comparison with Speculative Lock Elision

In Section 5.4.3 we discussed Speculative Lock Elision (SLE) as an additional method to exploit temporal silence. A detailed discussion of the ability of SLE and MESTI to capture temporal silence based on different program attributes can be found in our microbenchmark studies in Section 6.5.

We implement in core buffering to form speculative atomic regions; up to half of the ROB can be used for region creation to avoid sacrificing too much ILP when a proper idiom for SLE cannot be found. This is a change over performance results presented by Rajwar et al. [93, 90] where explicit store buffers and register checkpoints are used to create atomic regions. A detailed discussion of trade-offs between the two approaches can be found in Rajwar's thesis [90]. Practically speaking, as long as critical sections can fit within speculative buffering provided, any performance difference observed is due to second-order effects and not the SLE algorithm itself.

Furthermore, our implementation does not allow nested critical sections, as in the original SLE work [93]; in Rajwar's thesis [90] SLE was augmented to allow properly nested critical sections. Communication with the author of these works indicated that allowing nesting was not particularly beneficial for SLE [91]. Therefore, since handling nested critical sections with in core buffering presents non-trivial engineering complication, we did not implement it. We have shown in Section 6.5 that our SLE implementation faithfully captures the critical section idiom used in our microbenchmarks; the same idiom is used in our scientific workloads in the macros for the SPLASH-2 suite [107].

.

Finally, barriers to speculation in PHARMsim include: instruction and data TLB misses (speculative fills are not supported), improper alignment between loads and stores within the LSQ (the datapath cannot forward data between improperly aligned memory references), context synchronizing instructions (such as PowerPC isync), and reads of un-renamed registers (such as the machine status register). Some of these, e.g. load and store alignment and context synchronization, are discussed in Rajwar's thesis [90], but their impact is not quantified. In general, none of these limitations has proven consequential in practice; we note exceptions when necessary.

We show performance results for SLE compared to our enhanced MESTI implementation in Figure 6-16. All workloads are simulated statistically to enable concurrent critical section execution and performance measurement free from determinism-related effects. We do not present results for commercial workloads due to significant engineering challenges encountered in SLE implementation within PHARMsim for these workloads.

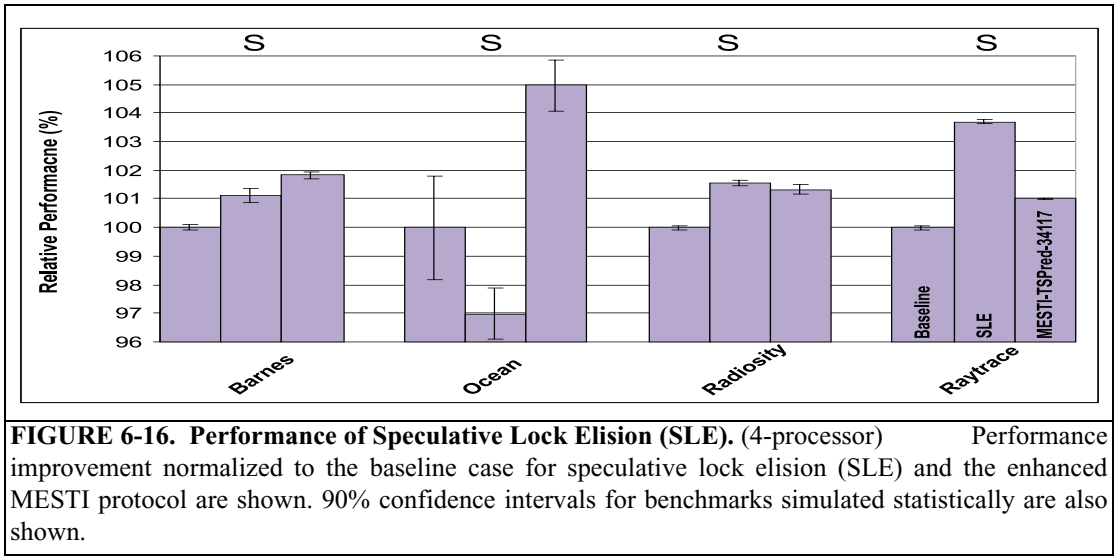

FIGURE 6-16. Performance of Speculative Lock Elision (SLE). (4-processor) Performance improvement normalized to the baseline case for speculative lock elision (SLE) and the enhanced MESTI protocol are shown. 90% confidence intervals for benchmarks simulated statistically are also shown.

We see that SLE achieves performance benefit in *barnes*, *radiosity*, and *raytrace* with speedup in *raytrace* being substantial. Comparing SLE performance to enhanced

MESTI, *barnes* and *radiosity* show essentially equivalent performance; SLE and MESTI effectively remove communication latency for temporally silent sharing patterns in these workloads. This indicates that critical sections are relatively uncontended in these workloads. *Raytrace* shows significant performance improvement for SLE over enhanced MESTI, indicating additional exposed concurrency through removing artificial dependences on the lock variable. This is discussed in detail in Section 6.5 with microbenchmark studies. *Ocean* shows significant performance degradation over both the baseline configuration and enhanced MESTI. Detailed examination of *ocean* revealed that excessive restarts and critical section conflicts are not the cause of slowdown, although they are a contributing factor. Rather, imprecision of the elision idiom is the primary culprit.

As discussed at length in Section 5.4.3, the load-locked/store-conditional pair which signals elision candidates may occur for many programming constructs in addition to spin-locks. In the case of *barnes*, *radiosity*, and *raytrace*, previous studies (Section 5.10.4) indicated most temporal silence occurs within application code, which conforms to the SLE idiom. However, *ocean* was a notable exception, exhibiting most temporal silence within the AIX kernel. Load-locked/store-conditional pairs occur frequently within the kernel for many purposes and therefore are not always amenable for elision.

Imprecision of the idiom leads to many false positives for elision candidates, causing speculative execution which does not lead to successful elision and therefore slows execution. Although our implementation attempts to recover from imprecise idioms as quickly as possible without performing a machine squash, and is highly successful at doing so in *ocean*, stalling memory operations within the processor core to attempt atomic region creation inherently slows program execution. Improving the idiom or adding

explicit instrumentation for lock acquires may mitigate this problem; we do not study such improvements in this thesis. In contrast, we note that MESTI performance is robust in the face of such issues, showing significant speedup in *ocean*.

In Figure 6-17, we show observed coherence transactions for the baseline, SLE, and enhanced MESTI. In all cases where SLE achieves performance improvement (*barnes*, *radiosity*, and *raytrace*), coherence transactions are successfully reduced; substantially in *raytrace*. In *ocean*, a slight increase is observed, indicating both SLE restarts and exclusive prefetches needed for atomic region creation may be slightly degrading performance [90]. However, as discussed, coherence interference is not the primary factor in the slowdown observed for *ocean* in Figure 6-16. Finally, note that although coherence transactions can be substantially reduced with SLE, a significant performance benefit does not come from this alone due to the low transaction rate in the baseline simulation; coherence bandwidth is sufficiently plentiful in our machine configuration for these workloads.

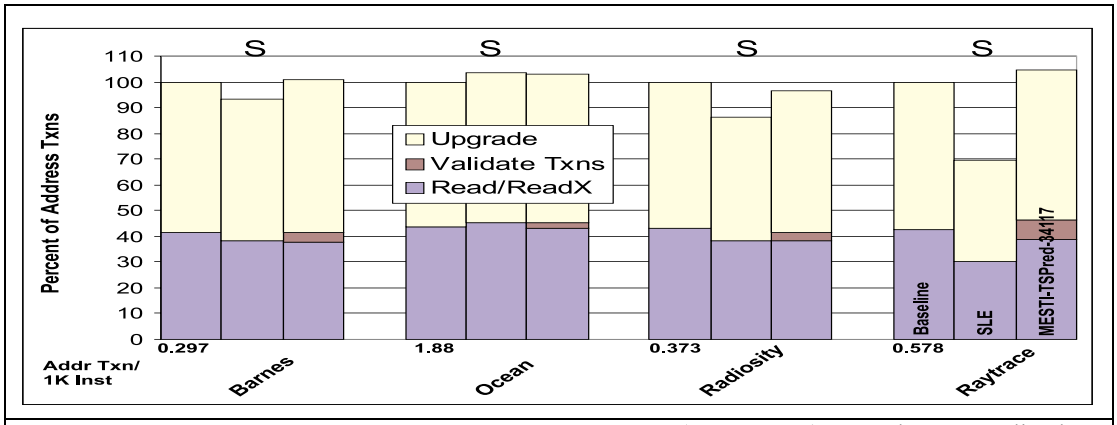

FIGURE 6-17. Address Transactions Observed with SLE. (4-processor) Results normalized to baseline case for speculative lock elision (SLE) and MESTI-TSPred-34117 (Section 5.7.3), are indicated. Address transaction rate per committed instruction is also shown.

6.7 Summary of Detailed Performance Evaluation

We introduced the PHARMsim simulation environment which allows simulation of both scientific and commercial workloads in a full-system, execution-driven, environ-

We used PHARMsim to present performance data for promising methods of exploiting store value locality, showing that exploiting update silent sharing (USS) can enable tangible performance benefit using the MESTI protocol. Substantial additional performance was accrued by eliminating temporal silent sharing (TSS) through the MESTI protocol. Adding simple methods of coherence prediction improved performance in both scientific and commercial workloads, showing the robustness of the MESTI protocol and the coherence prediction mechanism.

We also compared the non-speculative MESTI protocol against two speculative methods for taking advantage of TSS: speculative lock elision (SLE) and load value prediction with tag-match invalid cache lines (LVP). We compared the MESTI protocol with SLE in a detailed microbenchmark study, showing the performance potential of MESTI and SLE under ideal conditions, describing important program and machine parameters affecting performance of both approaches. Application benchmarks showed that all three approaches (MESTI, SLE, and LVP) can achieve tangible performance improvement, with detailed discussion of the strengths and weaknesses of each approach in different environments.

Chapter 7

Conclusion

This thesis shows that significant store value locality exists in programs, spanning single-threaded and multi-threaded programming paradigms, instruction set architectures, and compilation environments. Store value locality pervades program execution, and we can utilize it to improve uniprocessor and multiprocessor performance. We summarize contributions of the thesis and key results in Section 7.1 and discuss future research directions in Section 7.2.

7.1 Contributions and Summary of Results

This thesis illuminates that significant store value locality exists and can be utilized to improve memory system performance in uniprocessor and multiprocessor systems. We define *silent stores* as dynamic memory writes which exhibit exploitable store value locality. In this regard, we make two core contributions. We describe *update silent stores*, dynamic memory writes which contribute no change to system state because they write the value already existing at the memory location. We then expand the notion of store silence to *temporally silent stores* which revert the system state to a value observed previously.

7.1.1 Store Value Locality in Uniprocessors

Update silent stores can improve memory system performance in uniprocessors by increasing value transfer bandwidth and thereby reducing latency within the microprocessor core. We show that core efficiency can be improved and both on-chip and off-chip memory traffic can be effectively reduced via *update silent store suppression*, i.e. avoiding memory writes for update silent stores. Core efficiency is improved by reducing pressure

on microarchitectural structures, such as write buffers and cache ports. In write-through memory hierarchies, write-through memory traffic is reduced substantially; In writeback memory hierarchies many dirty castouts can be eliminated. We describe efficient methods of update silent store suppression, exploiting behavior of modern microprocessor cores and also ECC logic structure to enable suppression at very low cost.

We observed that the majority of performance improvement in microprocessor cores comes from reducing contention on existing structures for write handling (cache write ports, write buffers, etc.) Some fundamental ILP improvement is possible by eliminating true store to load dependences through memory on update silent stores, however this performance benefit is not studied in detail in this thesis as it was first proposed by others [111]. For core designs with substantial write handling bandwidth at all levels within the memory hierarchy we expect relatively little benefit. If update silent stores are exploited to remove true store to load dependences, we expect benefit to be proportional to the fraction of such dependences. Authors have shown a strong variability of such dependences on instruction set architecture (mostly correlated to the number of architected registers) and also instruction window size [111, 85]; therefore, potential benefit from this optimization will be strongly correlated to these aspects.

7.1.2 Update Silence in Multiprocessors

We make a fundamental contribution in defining *update silent sharing* (USS) to describe communication misses which are unnecessary due to update silent stores. We show that simple mechanisms considering USS can non-trivially reduce address and data transactions in multiprocessors through eliminating communication misses, writebacks, and invalidation messages. We show that naive update silent store suppression can be det-

rimental in multiprocessors when naively handling pure store misses with a Read transaction (to determine update silence) followed by an Upgrade for those which are not update silent. We define *critical update silence* to rigorously describe the phenomenon, and show simple mechanisms which enable effective critical update silence prediction.

Detailed performance studies with execution-driven, full-system simulation show USS enables tangible performance benefit in many programs. However, the set of communication misses which can be eliminated with USS is not substantial across all workloads we study, although many update silent stores (greater than 40% of dynamic stores) occur in virtually all workloads. This indicates that even though update silence pervades program execution, its participation in sharing misses (USS) is more algorithm-specific.

For those programs which benefit from exploiting USS, we perform a thorough investigation of implementation issues is modern architectures, effectively reducing our research to practice. We show, using the MESTI protocol, that tangible performance improvement can be obtained in workloads exhibiting USS while maintaining or slightly improving performance in other workloads. Of course, since one of the goals of USS is to eliminate communication misses and their associated latency to improve performance, the performance benefit is directly proportional to remote communication latency. Therefore, in systems where a large differential exists between local and remote memory accesses we expect greater benefit. In SMPs, the differential between local and remote accesses is large, and is expected to continue growing due to the memory-gap [46]—in future systems (such as CMPs) the differential may be reduced. However, the fundamental contribution of eliminating unnecessary communication in multiprocessors exists across all topologies.

We make a fundamental contribution in defining temporal silent sharing (TSS) to describe communication misses which return a value exactly matching the value previously observed by remote processors. These misses communicate no information affecting execution at remote processors since the memory value has reverted to the value last observed by the remote processor. We show that approximately 40% of communication misses in commercial workloads are TSS. We provide extensive characterization data to aid in designing efficient, general, mechanisms to exploit TSS not focused on any particular programming idiom. Additional characterization reveals numerous aspects of temporally silent program behavior, indicating that a substantial fraction of TSS occurs outside detectable synchronization primitives. This indicates that general mechanisms not focused on any particular programming idiom, such as spin-locks or critical sections, can be beneficial.

We show that a general, non-speculative, coherence protocol enhancement (MESTI) can capture a majority of TSS. We describe efficient engineering solutions to both detect and communicate TSS in current memory hierarchies and coherent interconnects. We introduce a speculative method which can also capture significant TSS—load value prediction with tag-match invalid cache lines (LVP). We compare these approaches to another speculative approach—Speculative Lock Elision (SLE). We show that all three approaches, MESTI, LVP, and SLE can achieve tangible performance benefit with a detailed discussion of the benefits and drawbacks of each.

These detailed studies lead to a few key findings. First, we observe that detecting temporal silence at the source, i.e. the processor creating the temporally silent pair, is

extremely beneficial since it can eliminate all remote miss latency for TSS cache lines, e.g. MESTI and SLE. Approaches which rely on speculation at the consumer, e.g. LVP, still expose substantial miss latency; this exposed latency can severely impact realized performance benefit even if a larger set of references can be captured.

Second, we observe that for well-tuned parallel programs with little conflict for critical sections, the MESTI protocol and SLE can achieve similar performance provided all TSS is only due to lock variables reliably indicated by SLE's idioms. If substantial TSS occurs outside idioms captured via SLE, MESTI can achieve greater performance since it does not target a specific programming idiom. For parallel programs with conflicting accesses to critical sections but few data conflicts, SLE can expose additional concurrency and also eliminate more communication misses leading to improved performance over MESTI. When comparing SLE and MESTI at a theoretical level, we observe that MESTI can only improve communication performance for a given memory reference pattern or program; SLE can eliminate communication specified in the program by removing unnecessary synchronization and exposing additional concurrency. In practice, we show many engineering reasons why either technique may be more or less desirable due to program/machine interactions and mechanisms used to implement either approach.

Third, we show that features of existing memory hierarchies, e.g. inclusive caches, ECC, etc., can be exploited to enable low-cost, general, detection of temporal silence for the MESTI protocol. In many cases, additional storage for stale memory versions can be obtained at only a slight bandwidth cost between the first and second levels in the memory hierarchy—since these two levels are, and will continue to be, on-chip for high performance processor designs, the slight bandwidth increase to reduce extensive chip-to-chip

communication may be a worthwhile trade-off.

Finally, we show that communicating useful temporal silence (only cases which prevent remote misses) is important in optimizing system performance when implementing the MESTI protocol. Simple coherence prediction structures which only observe physical addresses and require limited storage in the second level tag array can enable tangible performance benefit and virtually eliminate any performance degradation in machines implementing MESTI.

As for update silent stores in multiprocessors (Section 7.1.2), we propose temporal silence as an avenue to eliminate communication latency exposed to remote processors. Therefore, the benefit is related to communication latency within the system. Performance benefit obtained by exploiting TSS is therefore directly related to the relative cost of remote communication. However, the fundamental contribution of eliminating unnecessary communication in multiprocessors exists across all topologies.

7.2 Future Research Directions

We have shown that two simple types of store value locality, update silent stores and temporally silent stores, can substantially reduce unnecessary memory traffic. Researching additional dimensions of store value locality, utilizing the significant effort expended in other value prediction schemes, may further reduce memory traffic and unnecessary communication. Additional system topologies (such as directory-based coherence protocols, distributed shared memories, or multi-chip CMPs) can be also be explored to determine new engineering trade-offs. Finally, any system which relies on conservative memory ordering and disambiguation techniques to maintain correctness may be improved by considering store value locality. We have given examples in thread-

level speculation (TLS) systems as an illustration in Section 4.6.

On a broader level, a key result of this thesis is that significant computational energy is being expended to produce what amount to essentially useless memory values. We have already contributed much insight into this program behavior with this thesis and related work. Propagating this uselessness further up the dataflow, into operations contributing to these memory writes, may allow elimination or de-prioritization of computation to further improve performance delivered by uniprocessor and multiprocessor systems.

Beyond improving implementations through prioritization of computation or elimination of useless communication dynamically, we should also examine algorithms and dataflow specified in programs. The fact that approximately 40% of dynamic stores are update silent indicates a substantial inefficiency in instruction streams. This inefficiency may be rectified by programmers (simply through greater awareness and attention to algorithm design) or through improved, value sensitive, compilation techniques.

References

[1] D. Abts, D. J. Lilja, and S. Scott. Toward complexity-effective verification: A case study of the cray S{V2 cache coherence protocol, 2000.

[2] M. E. Acacio, J. Gonzalez, J. M. Garcia, and J. Duato. Use of prediction for accelerating upgrade misses in cc-NUMA multiprocessors. In *Proceedings of the International Conference on Parallel Architectures and Compilation Techniques*, September 2002.

[3] S. V. Adve and K. Gharachorloo. Shared memory consistency models: A tutorial. *IEEE Computer*, 29(12):66–76, December 1996.

[4] T. Agarwala and J. Cocke. High performance reduced instruction set processors. IBM Thomas J. Watson Research Center Technical Report #558845. March 1987.

[5] H. Akkary and M. A. Driscoll. A dynamic multithreading processor. In *Proceedings of the 31st Annual International Symposium on Microarchitecture*, pages 226–236, Dallas, TX, USA, 30 November–2 December 1998. ACM Press.

[6] A. R. Alameldeen, M. M. K. Martin, C. J. Mauer, K. E. Moore, M. Xu, D. J. Sorin, M. D. Hill, and D. A. Wood. Simulating a $2m commercial server on a $2k PC. *IEEE Computer*, 36(2):50–57, 2003.

[7] A. R. Alameldeen and D. A. Wood. Variability in architectural simulations of multi-threaded workloads. In *Proceedings of the 9th Annual International Symposium on High Performance Computer Architecture*, 2003.

[8] G. M. Amdahl. Validity of the single processor approach to achieving large scale computing capabilities. *AFIPS Proc. of the SJCC*, 31:483–485, 1967.

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

[10] G. B. Bell, K. M. Lepak, and M. H. Lipasti. A characterization of silent stores. In *Proceedings of the International Conference on Parallel Architectures and Compilation Techniques*, pages 133–142, Philadelphia, PA, October 2000.

[11] R. E. Blahut. *Theory and Practice of Error Control Codes*. Addison-Wesley Publishing Company, 1983.

[12] E. Borch, S. Manne, J. Emer, and E. Tune. Loose loops sink chips. In *Proc. of the 8th IEEE Symp. on High-Performance Computer Architecture (HPCA-8)*, 2002.

[13] J. Borkenhagen and S. Storino. 5th Generation 64-bit PowerPC-Compatible Commercial Processor Design. IBM Whitepaper available from http://www.rs6000.ibm.com, 1999.

[14] D.C. Burger and T.M. Austin. The simplescalar tool set, version 2.0. Technical report, University of Wisconsin Computer Sciences, 1997.

[15] H. W. Cain, K. M. Lepak, B. A. Schwartz, and M. H. Lipasti. Precise and accurate processor simulation. Proceedings of Computer Architecture Evaluation using Commercial Workloads (CAECW-02), February 2002.

[16] H. W. Cain and M. H. Lipasti. Verifying sequential consistency using vector clocks. In *Proceedings of the fourteenth annual ACM symposium on Parallel algorithms and architectures*, pages 153–154. ACM Press, 2002.

[17] H. W. Cain and M. H. Lipasti. Constraint graph analysis of multithreaded programs. In *Proceedings of the International Conference on Parallel Architectures and Compilation Techniques*, September 2003.

[18] H. W. Cain, R. Rajwar, M. Marden, and M. H. Lipasti. An architectural characterization of Java TPC-W. In *Proceedings of the Seventh International Symposium on High-Performance Computer Architecture*, pages 229–240, Monterrey, Mexico, January 2001.

[19] B. Calder, P. Feller, and A. Eustace. Value profiling. In *Proceedings of the 30th Annual ACM/IEEE International Symposium on Microarchitecture*, December 1997.

[20] B. Catanzaro. Multiprocessor system architectures: A technical survey of multiproces-

sor/multithreaded systems using PSPARC, multi-level bus architectures and Solaris (SunOS). Sun Microsystems, 1997.
[21] G. J. Chaitin, M.A. Auslander, A.K. Chandra, J. Cocke, M.E. Hopkins, and P.W. Markstein. Register allocation via coloring. *Computer Languages*, 6:47–57, 1981.
[22] A. Charlesworth, A. Phelps aand R. Williams, and G. Gilbert. Gigaplane-XB: Extending the ultra enterprise family. In *Proceedings of the International Symposium on High Performance Interconnects V*, August 1997.
[23] Y. Chen and M. Dubois. Cache protocol with partial block invalidation. In Proceedings of the 7th Int. Parallel Processing Symposium. IEEE Computer Society Press, 1993.
[24] M. Cintra and J. Torrellas. Eliminating squashes through learning cross-thread violations in speculative parallelization for multiprocessors. In *Proc. of the 8th IEEE Symp. on High-Performance Computer Architecture (HPCA-8)*, 2002.
[25] R. P. Colwell and R. Steck. A 0.6um BiCMOS processor with Dynamic Execution. In *Proceedings of ISSCC*, 1995.
[26] Compaq Computer Corporation. *Alpha 21264 Hardware Reference Manual DS-0027A-TE. http://www1.support.compaq.com/alpha-tools/documentation/current/chip-docs.html.*, 2000.
[27] A. Condon and A. J. Hu. Automatable verification of sequential consistency. In *ACM Symposium on Parallel Algorithms and Architectures*, pages 113–121, 2001.
[28] IBM Corporation. Fault tolerance decision in DRAM applications. Application Note, http://www.chips.ibm.com/products/memory/fault/fault.html, July 1997.
[29] IBM Corporation. AIX v4.3 online documentation. http://ncsp.upenn.edu/aix4.3html/, 2002.
[30] Intel Corporation. *Intel Pentium 4 Processor Optimization Reference Manual*. Intel Corporation, Santa Clara, CA, 2000.
[31] D. E. Culler and J. P. Singh. *Parallel Computer Architecture: A Hardware/Software Approach*. Morgan Kaufmann Publishers, Inc., San Mateo, CA, 1999.
[32] Advanced Micro Devices. AMD x86-64 architecture. Available from: http://www.x86-64.org.
[33] K. Diefendorff. K7 challenges Intel. *Microprocessor Report*, 12(7), October 1998.
[34] M. Dubois, L. Barroso, J. C. Wang, and Y. S. Chen. Delayed consistency and its effects on the miss rate of parallel programs. In Proceedings of Supercomputing ’91. ACM Press, 1991.
[35] M. Dubois, J. Skeppstedt, L. Ricciulli, K. Ramamurthy, and P. Stenström. The detection and elimination of useless misses in multiprocessors. In *20th Annual International Symposium on Computer Architecture*, May 1993.
[36] M. Dubois, J. Skeppstedt, and P. Strenstrom. Essential misses and data traffic in coherence protocols. *Journal of Parallel and Distributed Computing*, 29(2):108–125, 1995.
[37] D. T. Marr et. al. Hyper-Threading technology architecture and microarchitecture. *Intel Technology Journal*, 6(1), 2002.
[38] J. Ziegler et al. IBM experiments in soft fails in computer electronics. IBM Journal of Research and Development, January 1996.
[39] M. Franklin. *The Multiscalar Architecture*. PhD thesis, University of Wisconsin-Madison, 1993.
[40] K. Gharachorloo. *Memory Consistency Models for Shared-Memory Multiprocessors*. PhD thesis, Stanford University, 1995.
[41] K. Gharachorloo, M. Sharma, S. Steely, and S. Van Doren. Architecture and design of Alphaserver GS320, 2000.
[42] P. B. Gibbons and E. Korach. Testing shared memories. *SIAM Journal on Computing*, 26(4), 1997.
[43] J. R. Goodman and P. J. Woest. The wisconsin multicube: A new large-scale cache coherent multiprocessor. In *Proceedings of the 15th Annual International Symposium on Computer Architecture*, June 1988.

[44] H. Grahn and P. Stenström. Evaluation of a competitive-update cache coherence protocol with migratory data detection. *Journal of Parallel and Distributed Computing*, 39(2):168–180, 1996.
[45] A. Gupta and W. D. Weber. Cache invalidation patterns in shared-memory multiprocessors. *IEEE Transactions on Computers*, 41(7):794–810, July 1992.
[46] J.L Hennessy and D.A. Patterson. *Computer Architecture: A Quantitative Approach, 2nd Ed.*. Morgan Kaufmann Publishers, Inc., San Mateo, CA, 1996.
[47] Hewlett Packard Corporation. *PA-RISC 1.1 Architecture and Instruction Set Reference Manual. Third edition.*, 1994.
[48] M. D. Hill. *Aspects of cache memory and instruction buffer performance*. PhD thesis, The University of California at Berkeley, 1987.
[49] M. D. Hill. Multiprocessors should support simple memory consistency models. *IEEE Computer*, 31(8), August 1998.
[50] Z. Hu and S. Kaxiras. Timekeeping in the memory system: Predicting and optimizing memory behavior. In *Proceedings of the 29th International Symposium on Computer Architecture*, June 2002.
[51] Intel Corporation. *IA-64 Application Developer's Architecture Guide*, 1999.
[52] N. Jouppi. Cache write policies and performance. In *Proceedings of the 20th International Symposium on Computer Architecture*, San Diego, CA, 1993.
[53] S. Kaxiras and J. R. Goodman. Improving CC-NUMA performance using instruction-based prediction. In *Proceedings of the Fifth International Symposium on High-Performance Computer Architecture*, Orlando, January 1999.
[54] S. Kaxiras, Z. Hu, and M. Martonosi. Cache decay: Exploiting generational behavior to reduce cache leakage power. In *Proceedings of the 28th International Symposium on Computer Architecture*, June 2001.
[55] J. Keller. The 21264: A superscalar Alpha microprocessor with out-of-order execution. In *Proceedings of the Microprocessor Forum*, October 1996.
[56] T. Keller, A. M. Maynard, R. Simpson, and P. Bohrer. SimOS-PPC full system simulator. http://www.cs.utexas.edu/users/cart/simOS.
[57] I. Kim and M. H. Lipasti. Implementing optimizations at decode time. In *Proceedings of the 29th International Symposium on Computer Architecture*, pages 221–232, Anchorage, Alaska, May 2002.
[58] S. Kunkel, B. Armstrong, and P. Vitale. System optimization for OLTP workloads. *IEEE Micro*, May/June 1999.
[59] A. Lai and B. Falsafi. Memory sharing predictor: The key to a speculative coherent DSM. In *Proceedings of the 26th Annual International Symposium on Computer Architecture*, pages 172–183, 1999.
[60] A. Lai and B. Falsafi. Selective, accurate, and timely self-invalidation using last-touch prediction. In *Proceedings of the 27th Annual International Symposium on Computer Architecture*, pages 139–148, Vancouver, British Columbia, June 12–14, 2000.
[61] L. Lamport. How to make a multiprocessor computer that correctly executes multiprocess programs. *IEEE Transactions on Computers*, 28(9), 1979.
[62] A. Landin, E. Hagersten, and S. Haridi. Race-free interconnection networks and multiprocessor consistency. In *Proceedings of the 18th International Symposium on Computer Architecture (ISCA)*, 1991.
[63] G. Lauterbach and T. Horel. UltraSPARC-III: designing third generation 64-bit performance. *IEEE Micro*, 19(3):56–66, 1999.
[64] A. R. Lebeck and D. A. Wood. Dynamic self-invalidation: Reducing coherence overhead in shared-memory multiprocessors. In *Proceedings of the 22nd Annual International Symposium on Computer Architecture*, pages 48–59, 1995.
[65] H. Lee, G. Tyson, and M. Farrens. Eager writeback – a technique for improving bandwidth utilization. In *Proceedings of the 33rd ACM/IEEE International Symposium on Microarchitecture*, Monterrey, CA, November 2000.
[66] K. M. Lepak, G. B. Bell, and M. H. Lipasti. Silent stores and store value locality. *IEEE*

Transactions on Computers, 50(11), November 2001.
[67] K. M. Lepak, H. W. Cain, and M. H. Lipasti. Redeeming IPC as a performance metric for multithreaded programs. In *Proceedings of the International Conference on Parallel Architectures and Compilation Techniques*, September 2003.
[68] K. M. Lepak and M. H. Lipasti. On the value locality of store instructions. In *Proceedings of the 27th International Symposium on Computer Architecture*, pages 182–191, Vancouver, B.C., Canada, June 2000.
[69] K. M. Lepak and M. H. Lipasti. Silent stores for free. In *Proceedings of the 33rd ACM/IEEE International Symposium on Microarchitecture*, pages 22–31, Monterrey, CA, November 2000.
[70] K. M. Lepak and M. H. Lipasti. Temporally silent stores. In *Proceedings of the 10th Intl. Conf. on Architectural Support for Programming Languages and Operating Systems*, pages 30–41, October 2002.
[71] J. A. Lewis, B. Black, and M. H. Lipasti. Avoiding initialization misses to the heap. In *Proceedings of the 29th International Symposium on Computer Architecture*, pages 183–194, Anchorage, Alaska, May 2002.
[72] M. H. Lipasti and J. P. Shen. Exceeding the dataflow limit via value prediction. In *Proceedings of the 29th Annual ACM/IEEE International Symposium on Microarchitecture*, December 1996.
[73] M. H. Lipasti, C. B. Wilkerson, and J. P. Shen. Value locality and load value prediction. In *Proceedings of the Seventh International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS-VII)*, October 1996.
[74] M. M. K. Martin, P. Harper, D. Sorin, M. Hill, and D. Wood. Using destination-set prediction to improve the latency/bandwidth tradeoff in shared-memory multiprocessors. In *Proceedings of the 30th International Symposium on Computer Architecture*, pages 206–217, San Diego, CA, U.S.A., June 2003.
[75] M. M. K. Martin, D. J. Sorin, A. Ailamaki, A. R. Alameldeen, R. M. Dickson, C. J. Mauer, K. E. Moore, M. Plakal, M. D. Hill, and D. A. Wood. Timestamp snooping: An approach for extending SMPs. *ACM SIG-PLAN Notices*, 35(11):25–36, November 2000.
[76] M. M. K. Martin, D. J. Sorin, H. W. Cain, M. D. Hill, and M. H. Lipasti. Correctly implementing value prediction in microprocessors that support multithreading or multiprocessing. In *Proceedings of MICRO-34*, December 2001.
[77] J. F. Martinez and J. Torrellas. Speculative synchronization: Applying thread-level speculation to explicitly parallel applicatioons. In *Proceedings of the 10th Intl. Conf. on Architectural Support for Programming Languages and Operating Systems*, October 2002.
[78] C. May, E. Silha, R. Simpson, and H. Warren. *The PowerPC Architecture*. Morgan Kaufmann Publishers, Inc., 1994.
[79] T. May and M. Woods. Alpha-particle-induced soft errors in dynamic memories. IEEE Transactions on Electronic Devices, 1979.
[80] C. McNairy and D. Soltis. Itanium 2 processor microarchitecture. *IEEE Micro*, 23(2):44–55, 2003.
[81] A. Mendelson and F. Gabbay. Speculative execution based on value prediction. Technical report, Technion, 1997. (http://www-ee.technion.ac.il/%7efredg).
[82] C. Molina, A. Gonzalez, and J. Tubella. Reducing memory traffic via redundant store instructions. In *Proc. of Int. Conf. on High Perf. Computing and Networking*, pages 1246–1249, April 1999.
[83] C. Moore. POWER4 system microarchitecture. In *Proceedings of the Microprocessor Forum*, October 2000.
[84] G. E. Moore. Cramming more components onto integrated circuits. *Electronics*, April 1965.
[85] A. Moshovos. *Memory Dependence Prediction*. PhD thesis, University of Wisconsin, 1998.

[86] S. S. Mukherjee and M. D. Hill. Using prediction to accelerate coherence protocols. In *Proceedings of the 25th International Symposium on Computer Architecture*, pages 179–190, Barcelona, Spain, June 1998.
[87] J. Nilsson and F. Dahlgren. Improving performance of load-store sequences for transaction processing workloads on multiprocessors. In *Proceedings of the International Conference on Parallel Processing*, Sept 1999.
[88] J. Nilsson and F. Dahlgren. Reducing ownership overhead for load-store sequences in cache-coherent multiprocessors. In *Proceedings of the 2000 International Parallel and Distributed Processing Symposium*, May 2000.
[89] V. Pai, P. Ranganathan, and S. Adve. Rsim: A simulator for shared-memory multiprocessor and uniprocessor systems that exploit ILP. In Proceedings of the 3rd Workshop on Computer Architecture Education, 1997.
[90] R. Rajwar. *Speculation-Based Techniques for Transactional Lock-Free Execution of Lock-Based Programs*. PhD thesis, University of Wisconsin, 2002.
[91] R. Rajwar. Personal Communication, August 2003.
[92] R. Rajwar and J. R. Goodman. SimpleMP multiprocessor simulator. Personal communication, 2000.
[93] R. Rajwar and J. R. Goodman. Speculative lock elision: Enabling highly concurrent multithreaded execution. In *Proceedings of the 34th ACM/IEEE International Symposium on Microarchitecture*, pages 294–305, December 2001.
[94] R. Rajwar and J. R. Goodman. Transactional lock-free execution of lock-based programs. In *Proceedings of the 10th Intl. Conf. on Architectural Support for Programming Languages and Operating Systems*, October 2002.
[95] A. Ramirez, L. A. Barroso, K. Gharachorloo, R. Cohn, J.-L. Larriba-Pey, P. G. Lowney, and M. Valero. Code layout optimizations for transaction processing workloads. In *Proceedings of the 28th International Symposium on Computer Architecture*, June 2001.
[96] P. Ranganathan, K. Gharachorloo, S. V. Adve, and L. A. Barroso. Performance of database workloads on shared-memory systems with out-of-order processors. In *Proceedings of the 8th Intl. Conf. on Architectural Support for Programming Languages and Operating Systems*, San Jose, CA, October 1998.
[97] T. R. N. Rao and E. Fujiwara. *Error-Control Coding for Computer Systems*. Prentice Hall, Englewood Cliffs, NJ, 1989.
[98] M. Raynal. *Algorithms for Mutual Exclusion*. The MIT Press, 1986.
[99] M. Rosenblum. SimOS full system simulator. http://simos.stanford.edu.
[100] Eric Rotenberg. Ar-smt: A microarchitectural approach to fault tolerance in microprocessors. In *Proceedings of the 29th Fault-Tolerant Computing Symposium*, June 1999.
[101] P. Rubinfeld. Managing problems at high speed. IEEE Computer, January 1998.
[102] J. E. Smith. A study of branch prediction techniques. In *Proceedings of the 8th Annual Symposium on Computer Architecture*, pages 135–147, June 1981.
[103] J. G. Steffan, C. B. Colohan, A. Zhai, and T. C. Mowry. Improving value communication for thread-level speculation. In *Proc. of the 8th IEEE Symp. on High-Performance Computer Architecture (HPCA-8)*, 2002.
[104] K. Sundaramoorthy, Z. Purser, and E. Rotenberg. Slipstream processors: Improving both performance and fault tolerance. In *Architectural Support for Programming Languages and Operating Systems*, pages 257–268, 2000.
[105] J. M. Tendler, J. S. Dodson, J. S. Fields, H. Le, and B. Sinharoy. POWER4 system microarchitecture. http://www-1.ibm.com/servers/eserver/pseries/hardware/whitepapers/power4.htm% l, November 2001.
[106] Transaction Processing Performance Council. TPC benchmarks. http://www.tpc.org.
[107] S. C. Woo, M. Ohara, E. Torrie, J. P. Singh, and A. Gupta. The SPLASH-2 programs: Characterization and methodological considerations. In *Proceedings of the 22nd International Symposium on Computer Architecture*, June 1995.

[108] D. A. Wood, M. D. Hill, and R. E. Kessler. A model for estimating trace-sample miss ratios. In *Proceedings of the 1991 ACM Sigmetrics Conference on Measurement and Modeling of Computer Systems*, 1991.

[109] J. Yang and R. Gupta. Energy-efficient load and store reuse. In *IEEE/ACM International Symposium on Low Power Electronics and Design (ISLPED)*, 2001.

[110] K. C. Yeager. The MIPS R10000 superscalar microprocessor. *IEEE Micro*, April 1996.

[111] A. Yoaz, R. Ronen, R. S. Chappell, and Y. Almog. Silence is golden? In Proceedings of Work-in-Progress Workshop in conjunction with HPCA-7, January 2001.

[112] J. Ziegler. Terrestrial cosmic rays. IBM Journal of Research and Development, January 1996.

[113] C. Zilles and G. Sohi. Master/Slave Speculative Parallelization. In *Proc. of the 35th International Symposium on Microarchitecture (Micro-35)*, November 2002.

Appendix A.

MOESTI Protocol

A.1 Verification of the MOESTI Protocol and Implementation Considerations

We have discussed general implementation issues in the MESTI protocol in Section 5.8.3. Protocols used in commercial implementations can be significantly more complicated. For example, our detailed simulation environment, PHARMsim, implements a variation of the Gigaplane-XB protocol, with three major state machines (in the L1 cache, L2 cache, and DTAG) collaborating to ensure correct operation. We have discussed this previously in Section 6.2.2. In this section, we discuss major differences between the actual MOESTI protocol implemented and previous MESTI descriptions, details of update silent store handling within the protocol, and conclude with a brief discussion of correctness verification.

A.1.1 MOESTI Correctness Considerations

The largest single difference for MOESTI as compared to MESTI is the obvious addition of the Owned state. In our protocol, Owned implies a cache line is dirty with respect to memory but that it is potentially shared by other processors in the system. Therefore, if an Owned line is cast out, it must be written back to memory; however Owned does not imply the cache line is only cached by the owning processor.

As discussed in Section 5.8.3, in basic MESTI any validate transaction should transfer ownership for the cache block back to main memory to ensure correct protocol operation. As long as the protocol is correctly implemented, no data transfer is required for this *implicit writeback* because validated data matches the contents of memory. In MOESTI, memory data can be incorrect with respect to validated data due to the Owned

state. Therefore, any time a validate is broadcast to the system (and thus the validating processor desires to downgrade from Modified state), it must return to the Owned state to ensure correct protocol operation, implying that any validated cache line may also lead to a main memory writeback. This may not be strictly required if the implementation tracks the previous state of the cache line prior to entering Modified state (Exclusive, Shared) and conditionally returns to a *clean* state; however, in a Gigaplane-XB-like implementation this optimization is not possible. The reason is subtle and difficult to explain for those not intimately familiar with Gigaplane-XB. At a high level, ownership, i.e. data sourcing responsibility to remote requestors, is determined at the DTAG when snoop-requests and snoop-responses are generated; once a processor has claimed ownership of a cache block and asserted the appropriate snoop-response, it must supply data to the requestor. Therefore, to handle a race case when an external request is in-flight between the DTAG (and therefore has had ownership claimed by the validating processor) and the L2 for validated data, the L2 must return to the Owned state (instead of E or S) to ensure it will correctly source the data to an external requestor.

Another subtlety arises in correctly assuring that validates are associated with the proper temporally invalid (T) state and validating processor. We have described this in Section 5.8.3 as assuring *correct correspondence* between T state and validating processors. In our Gigaplane-XB-like protocol, correct correspondence is assured at the DTAG by an owning processor (in Owned state at the DTAG) asserting the *ignore* snoop-response to any validate which it did not broadcast. This causes no forward progress or multiple bus-driver issues; forward progress is ensured because eliminating validates in MESTI is always correct (Section 5.8.3), and single assertion of *ignore* is assured because

only one processor can be in the Owned state at the DTAG by protocol design. Elsewhere within the coherence hierarchy, e.g. the delay queue discussed in Section 6.6.2, correct correspondence is assured by snooping request/response queues appropriately between levels in the memory hierarchy for external requests.

A.1.2 Update Silent Store Handling in PHARMsim and MOESTI

We discussed a general mechanism for handling update silent store clean hits and pure store misses using the MESTI protocol in Section 5.7.4 with many desirable properties. We implement this mechanism in PHARMsim. Furthermore, we extend it slightly to reduce coherence transaction latency in the case of pure store misses (store misses which hit an invalid state and therefore no data is available to verify update silence). We assume that store data for the store causing the miss can be transferred to the L2 cache, or is available as part of the L2 miss status holding register (MSHR) to enable comparison with incoming coherence network data when it hits the L2. This comparison allows the L2 to determine immediately whether the store is update silent, leading to two possible outcomes: In the case of update silence, the L2 installs the cache line in Owned state, adds a validate to the outbound coherence queue, and responds to the L1 cache informing it of update silence and required installation in Shared state when the data reaches the L1. In the case of non-update silence, the L2 installs the cache line in Modified state and responds to the L1 cache with the data leading to installation in Modified state. This is baseline coherence operation.

If update silence is not verified at the L2 level, the latency for validate broadcast is increased by two queuing delays incurred through the inbound and outbound queues between the L1 and L2 caches. It also leads to additional transitions at the L2 level for

each update silent store miss, increasing occupancy and potentially power consumption. However, either implementation is correct and possible to implement.

A.1.3 MOESTI Correctness Verification

The MOESTI protocol was verified by implementing it in PHARMsim and utilizing the verification infrastructure described in Section 6.3.1. Extensive application benchmark testing and microbenchmark testing with instruction-by-instruction verification using a semantically unmodified SimOS-PPC has given us high confidence of the correctness of our implementation.

As discussed in Section 4.5.2, a potential correctness issue arises under PowerPC for update silent store suppression in the case of update silent store-clean hits (an update silent store hits Shared or Owned state, and thus no upgrade is generated) because of side-effects on the reservation register used to implement load-locked/store-conditional operations. We showed there that suppression can be observed in PowerPC under this scenario and a deviation in program execution can result under contrived circumstances. Our verification infrastructure was augmented to detect when true sharing on the reservation address occurred and update silent store suppression to the same address caused a deviation in execution; we observed no execution errors in any benchmarks or test programs.

A.2 Detailed Description of the MOESTI Protocol Used in PHARMsim

In Section A.1 we have described the MOESTI implementation issues inside our performance simulator, PHARMsim and high-level changes to the baseline MOESI protocol which must be considered. What follows in this section are detailed protocol state machine descriptions (with only states and transitions, internal events have been removed for clarity) for both the DTAG state machine and the L2 state machine. The MOESTI

DTAG implements four states: Owned, Shared, T_invalid, and Invalid. The base PHARMsim MOESI DTAG implements three states: Owned, Shared, and Invalid. The addition of T_invalid at the DTAG is necessary to enable proper snoop-filtering for the L2 state machine; simply routing all transactions from Invalid state to the L2 requires additional state/transition pairs in the L2 state machine and no external snoops can be filtered to maintain correct MESTI operation.

The MOESTI L2 state machine implements eight stable states: Modified, Modified_Clean, Owned, Exclusive, Shared, Validate_Shared, T_Invalid, and Invalid and an additional 39 transient states. The base PHARMsim MOESI L2 state machine implements six stable states: Modified, Modified_Clean, Owned, Exclusive, Shared, and Invalid and an additional 37 transient states. The added stable states are most relevant, and we have described their utility in detail in Chapter 5 along with general correctness considerations for implementing MESTI/MOESTI. We have not made a substantial effort to reduce the number of transient states in our simulation environment. The state machine is coded in C; the descriptions that follow have been extracted from the protocol C code using an automated extraction tool. The descriptions are formatted for visualization with DOT/DOTTY which is part of the AT&T graphviz suite available from http://www.research.att.com/sw/tools/graphviz/ as of this writing.[1] The representation essentially describes a directed graph with nodes being stable and transient states and arcs being transitions.

1. Previously, we had included graphical output from dotty. However, since the number of pages was excessive (and not particularly useful in our opinion), we have chosen to express each state machine in the DOT format. The user can input the descriptions into dotty for visualization, if desired.

```
digraph "State transition diagram for l2_dtag"
{
page="8.5,11"
margin="1.25"
size="25.5,18"
rotate=90
ratio=auto
node
[shape=circle,style=filled,fontname=Helvetica,fontsize=12,height=.1];
edge [fontname=Helvetica,fontsize=12];
0           [label="dtag_invalid"];
1           [label="dtag_t_invalid"];
2           [label="dtag_shared"];
3           [label="dtag_owned"];
0           ->          0           [label="validate_self"];
1           ->          1           [label="validate_self"];
0           ->          0           [label="validate_other"];
1           ->          2           [label="validate_other"];
0           ->          3           [label="rd_self_not_shared"];
0           ->          0           [label="ignore"];
1           ->          1           [label="ignore"];
2           ->          2           [label="ignore"];
3           ->          3           [label="ignore"];
0           ->          2           [label="rd_self_shared"];
1           ->          3           [label="rd_self"];
0           ->          2           [label="rd_inst_self"];
1           ->          2           [label="rd_inst_self"];
0           ->          3           [label="rfo_self"];
1           ->          3           [label="rfo_self"];
0           ->          3           [label="upg_self"];
1           ->          3           [label="upg_self"];
0           ->          0           [label="upg_stc_self \l-
assert_ignore"];
1           ->          0           [label="upg_stc_self \l-
assert_ignore"];
0           ->          0           [label="wb_self \l-assert_ignore"];
1           ->          0           [label="wb_self \l-assert_ignore"];
0           ->          0           [label="rd_other"];
0           ->          0           [label="rd_inst_other"];
0           ->          0           [label="wb_other"];
0           ->          0           [label="rfo_other"];
0           ->          0           [label="upg_other"];
0           ->          0           [label="upg_stc_other"];
0           ->          0           [label="dcb_zero_other"];
1           ->          0           [label="rd_other"];
1           ->          0           [label="rd_inst_other"];
1           ->          0           [label="rfo_other"];
1           ->          0           [label="upg_other"];
1           ->          0           [label="upg_stc_other"];
1           ->          0           [label="dcb_zero_other"];
1           ->          1           [label="wb_other"];
0           ->          3           [label="dcb_zero_self"];
1           ->          3           [label="dcb_zero_self"];
0           ->          0           [label="icb_inval_self"];
```

```
1           ->          1           [label="icb_inval_self"];
0           ->          0           [label="dcb_flush_self"];
0           ->          0           [label="dcb_inval_self"];
1           ->          0           [label="dcb_flush_self"];
1           ->          0           [label="dcb_inval_self"];
0           ->          0           [label="dcb_zero_other"];
0           ->          0           [label="dcb_flush_other"];
0           ->          0           [label="dcb_inval_other"];
1           ->          0           [label="dcb_flush_other"];
1           ->          0           [label="dcb_inval_other"];
0           ->          0           [label="icb_inval_other"];
1           ->          0           [label="icb_inval_other"];
2           ->          2           [label="validate_self"];
2           ->          2           [label="validate_other"];
2           ->          2           [label="rd_self"];
2           ->          3           [label="rfo_self"];
2           ->          3           [label="upg_self"];
2           ->          3           [label="upg_stc_self"];
2           ->          2           [label="wb_self"];
2           ->          2           [label="rd_other"];
2           ->          2           [label="rd_inst_other"];
2           ->          1           [label="rfo_other"];
2           ->          1           [label="upg_other"];
2           ->          1           [label="upg_stc_other"];
2           ->          2           [label="wb_other"];
2           ->          3           [label="dcb_zero_self"];
2           ->          0           [label="dcb_flush_self"];
2           ->          0           [label="dcb_inval_self"];
2           ->          2           [label="icb_inval_self"];
2           ->          1           [label="dcb_zero_other"];
2           ->          1           [label="dcb_flush_other"];
2           ->          1           [label="dcb_inval_other"];
2           ->          2           [label="icb_inval_other"];
3           ->          3           [label="validate_self"];
3           ->          3           [label="validate_other \l-
assert_ignore"];
3           ->          3           [label="rd_self"];
3           ->          3           [label="rfo_self"];
3           ->          3           [label="upg_self"];
3           ->          3           [label="upg_stc_self"];
3           ->          0           [label="wb_self"];
3           ->          3           [label="rd_inst_other"];
3           ->          3           [label="rd_other"];
3           ->          1           [label="rfo_other"];
3           ->          1           [label="upg_other"];
3           ->          1           [label="upg_stc_other"];
3           ->          3           [label="wb_other"];
3           ->          3           [label="dcb_zero_self"];
3           ->          0           [label="dcb_flush_self"];
3           ->          0           [label="dcb_inval_self"];
3           ->          3           [label="icb_inval_self"];
3           ->          1           [label="dcb_zero_other"];
3           ->          1           [label="dcb_inval_other"];
3           ->          3           [label="dcb_flush_other"];
```

```
3               ->              3               [label="icb_inval_other"];
}
```

```
digraph "State transition diagram for l2_with_MOESTI"
{
page="8.5,11"
margin="1.25"
size="24,34"
#size="6,8.5"
ratio=auto
rotate=90
node
[shape=circle,style=filled,fontname=Helvetica,fontsize=11,height=.1];
edge [fontname=Helvetica,fontsize=11];
0           [label="t_invalid"];
1           [label="invalid"];
2           [label="exclusive"];
3           [label="shared"];
4           [label="val_shared"];
5           [label="modified"];
6           [label="modified_clean"];
7           [label="owned"];
8           [label="pn_i\l_none_e"];
9           [label="pn_i_e\l_d_na"];
10          [label="pn_i_e\l_nd_a"];
11          [label="pn_i_e_a\l_xdcbf_nd"];
12          [label="pn_i_e_a\l_xrd_nd"];
13          [label="pn_i_e_a\l_xrfo_xrd_nd"];
14          [label="pn_i_e_a\l_xrfo_nd"];
15          [label="pn_i\l_none_s"];
16          [label="pn_i_s\l_d_na"];
17          [label="pn_i_s\l_nd_a"];
18          [label="pn_i_s_a\l_xdcbf_nd"];
19          [label="pn_i_s_a\l_xrfo_nd"];
20          [label="pn_i\l_none_m"];
21          [label="pn_s_m\l_xrfo_none"];
22          [label="pn_o_m\l_xrfo_none"];
23          [label="pn_i_m\l_d_na"];
24          [label="pn_i_m\l_nd_a"];
25          [label="pn_i_m_a\l_xrfo_nd"];
26          [label="pn_i_m_a\l_xrd_nd"];
27          [label="pn_i_m_a\l_xrfo_xrd_nd"];
28          [label="pn_s\l_none_m"];
29          [label="pn_s_m\l_none_stc"];
30          [label="pn_s_i\l_none_stc"];
31          [label="pn_o\l_none_m"];
32          [label="pn_o_m\l_none_stc"];
33          [label="pn_m\l_none_i"];
34          [label="pn_m\l_dcbf_i"];
35          [label="pn_m_i\l_xrfo_dcbf"];
36          [label="pn_m_i\l_xrfo_none"];
37          [label="pn_o\l_none_i"];
38          [label="pn_m\l_flush_o"];
39          [label="pn_m\l_flush_i"];
40          [label="pn_e\l_none_i"];
41          [label="pn_i\l_icbi_i"];
42          [label="pn_s\l_icbi_s"];
```

```
43          [label="pn_m\l_icbi_m"];
44          [label="pn_mc\l_icbi_mc"];
45          [label="pn_e\l_icbi_e"];
46          [label="pn_o\l_icbi_o"];
0           ->          4           [label="validate_other"];
0           ->          8           [label="prefetch"];
0           ->          8           [label="rd"];
0           ->          20          [label="rfo"];
0           ->          20          [label="prefetch_x"];
0           ->          20          [label="upg"];
0           ->          1           [label="upg_stc"];
0           ->          15          [label="rd_inst"];
0           ->          1           [label="rd_other"];
0           ->          1           [label="rd_inst_other"];
0           ->          0           [label="wb_other"];
0           ->          1           [label="rfo_other"];
0           ->          1           [label="upg_other"];
0           ->          1           [label="upg_stc_other"];
0           ->          1           [label="l1_rsp_rd_other"];
0           ->          1           [label="l1_rsp_rfo_other"];
0           ->          1           [label="l1_rsp_upg_other"];
0           ->          1           [label="l1_rsp_evict"];
0           ->          20          [label="dcb_zero"];
0           ->          20          [label="dcb_flush"];
0           ->          41          [label="icb_inval"];
0           ->          20          [label="dcb_inval"];
0           ->          1           [label="icb_inval_other"];
0           ->          1           [label="icb_inval_self"];
0           ->          1           [label="dcb_zero_other"];
0           ->          1           [label="dcb_flush_other"];
0           ->          1           [label="dcb_inval_other"];
0           ->          1           [label="dcb_inval_self"];
1           ->          1           [label="l1_wb"];
1           ->          1           [label="validate_self"];
1           ->          8           [label="prefetch"];
1           ->          8           [label="rd"];
1           ->          20          [label="rfo"];
1           ->          20          [label="prefetch_x"];
1           ->          20          [label="upg"];
1           ->          1           [label="upg_stc"];
1           ->          15          [label="rd_inst"];
1           ->          1           [label="rd_other"];
1           ->          1           [label="rd_inst_other"];
1           ->          1           [label="wb_other"];
1           ->          1           [label="rfo_other"];
1           ->          1           [label="upg_other"];
1           ->          1           [label="upg_stc_other"];
1           ->          1           [label="l1_rsp_rd_other"];
1           ->          1           [label="l1_rsp_rfo_other"];
1           ->          1           [label="l1_rsp_upg_other"];
1           ->          1           [label="l1_rsp_evict"];
1           ->          20          [label="dcb_zero"];
1           ->          20          [label="dcb_flush"];
1           ->          41          [label="icb_inval"];
```

```
1       ->      20      [label="dcb_inval"];
1       ->      1       [label="icb_inval_other"];
1       ->      1       [label="icb_inval_self"];
1       ->      1       [label="dcb_zero_other"];
1       ->      1       [label="dcb_flush_other"];
1       ->      1       [label="dcb_inval_other"];
1       ->      1       [label="dcb_inval_self"];
2       ->      2       [label="l1_wb"];
2       ->      2       [label="rd"];
2       ->      5       [label="rfo"];
2       ->      2       [label="prefetch"];
2       ->      2       [label="prefetch_x"];
2       ->      5       [label="upg"];
2       ->      5       [label="upg_stc"];
2       ->      2       [label="rd_inst"];
2       ->      40      [label="repl"];
2       ->      7       [label="rd_inst_other"];
2       ->      7       [label="rd_other"];
2       ->      0       [label="rfo_other"];
2       ->      5       [label="dcb_zero"];
2       ->      37      [label="dcb_flush"];
2       ->      45      [label="icb_inval"];
2       ->      37      [label="dcb_inval"];
2       ->      0       [label="dcb_zero_other"];
2       ->      0       [label="dcb_flush_other"];
2       ->      0       [label="dcb_inval_other"];
2       ->      2       [label="icb_inval_other"];
3       ->      3       [label="l1_wb"];
4       ->      4       [label="l1_wb"];
3       ->      3       [label="rd"];
4       ->      3       [label="rd"];
3       ->      3       [label="prefetch"];
4       ->      3       [label="prefetch"];
3       ->      28      [label="rfo_non_silent"];
3       ->      3       [label="rfo_silent"];
3       ->      3       [label="prefetch_x_silent"];
3       ->      28      [label="prefetch_x_non_silent"];
4       ->      3       [label="rfo_silent"];
3       ->      28      [label="rfo_non_silent"];
4       ->      3       [label="prefetch_x_silent"];
3       ->      28      [label="prefetch_x_non_silent"];
3       ->      28      [label="upg_non_silent"];
3       ->      3       [label="upg_silent"];
4       ->      28      [label="upg_non_silent"];
4       ->      3       [label="upg_silent"];
3       ->      29      [label="upg_stc"];
4       ->      29      [label="upg_stc"];
3       ->      3       [label="rd_inst"];
4       ->      3       [label="rd_inst"];
3       ->      1       [label="repl"];
4       ->      1       [label="repl"];
3       ->      3       [label="rd_other"];
4       ->      3       [label="rd_other"];
3       ->      0       [label="rfo_other"];
```

```
4           ->          0           [label="rfo_other"];
3           ->          0           [label="upg_other"];
4           ->          0           [label="upg_other"];
3           ->          0           [label="upg_stc_other"];
4           ->          0           [label="upg_stc_other"];
3           ->          0           [label="dcb_zero_other"];
3           ->          0           [label="dcb_flush_other"];
3           ->          0           [label="dcb_inval_other"];
4           ->          0           [label="dcb_zero_other"];
4           ->          0           [label="dcb_flush_other"];
4           ->          0           [label="dcb_inval_other"];
3           ->          3           [label="icb_inval_other"];
4           ->          4           [label="icb_inval_other"];
3           ->          20          [label="dcb_zero"];
4           ->          20          [label="dcb_zero"];
3           ->          20          [label="dcb_flush"];
4           ->          20          [label="dcb_flush"];
3           ->          42          [label="icb_inval"];
4           ->          42          [label="icb_inval"];
3           ->          20          [label="dcb_inval"];
4           ->          20          [label="dcb_inval"];
5           ->          39          [label="repl"];
5           ->          6           [label="l1_wb"];
5           ->          6           [label="l1_wb_nots"];
5           ->          5
[label="validate_nobcast_reupgrade"];
5           ->          6
[label="validate_nobcast_noreupgrade"];
5           ->          7           [label="validate_bcast"];
5           ->          38          [label="rd_inst_other"];
5           ->          38          [label="rd_other"];
5           ->          39          [label="rfo_other"];
5           ->          39          [label="upg_other"];
5           ->          43          [label="dcb_zero"];
5           ->          43          [label="dcb_flush"];
5           ->          43          [label="icb_inval"];
5           ->          37          [label="dcb_inval"];
5           ->          0           [label="dcb_zero_other"];
5           ->          39          [label="dcb_flush_other"];
5           ->          5           [label="icb_inval_other"];
5           ->          5           [label="icb_inval_self"];
5           ->          1           [label="dcb_inval_other"];
5           ->          38          [label="rd_inst"];
6           ->          6           [label="l1_wb"];
6           ->          6           [label="rd_inst"];
6           ->          6           [label="rd"];
6           ->          6           [label="prefetch"];
6           ->          6           [label="prefetch_x"];
6           ->          5           [label="rfo"];
6           ->          5           [label="upg"];
6           ->          5           [label="upg_stc"];
6           ->          33          [label="repl"];
6           ->          7           [label="rd_inst_other"];
6           ->          7           [label="rd_other"];
```

```
6       ->      0       [label="rfo_other"];
6       ->      0       [label="upg_other"];
6       ->      5       [label="dcb_zero"];
6       ->      34      [label="dcb_flush"];
6       ->      44      [label="icb_inval"];
6       ->      37      [label="dcb_inval"];
6       ->      0       [label="dcb_zero_other"];
6       ->      33      [label="dcb_flush_other"];
6       ->      6       [label="icb_inval_other"];
6       ->      0       [label="dcb_inval_other"];
7       ->      7       [label="l1_wb"];
7       ->      7       [label="rd"];
7       ->      7       [label="rd_inst"];
7       ->      7       [label="prefetch"];
7       ->      7       [label="rfo"];
7       ->      7       [label="prefetch_x"];
7       ->      7       [label="upg"];
7       ->      7       [label="upg_stc"];
7       ->      37      [label="repl"];
7       ->      7       [label="rd_inst_other"];
7       ->      7       [label="rd_other"];
7       ->      0       [label="rfo_other"];
7       ->      0       [label="upg_other"];
7       ->      0       [label="upg_stc_other"];
7       ->      7       [label="dcb_zero"];
7       ->      34      [label="dcb_flush"];
7       ->      46      [label="icb_inval"];
7       ->      37      [label="dcb_inval"];
7       ->      0       [label="dcb_zero_other"];
7       ->      37      [label="dcb_flush_other"];
7       ->      7       [label="icb_inval_other"];
7       ->      0       [label="dcb_inval_other"];
8       ->      8       [label="rfo_other"];
8       ->      8       [label="upg_other"];
8       ->      8       [label="upg_stc_other"];
8       ->      8       [label="dcb_zero_other"];
8       ->      8       [label="rd_other"];
8       ->      8       [label="rd_inst_other"];
8       ->      8       [label="wb_other"];
8       ->      8       [label="dcb_flush_other"];
8       ->      8       [label="icb_inval_other"];
8       ->      8       [label="dcb_inval_other"];
8       ->      10      [label="rd_self"];
8       ->      9       [label="dat"];
9       ->      2       [label="rd_self"];
10      ->      12      [label="rd_other"];
10      ->      12      [label="rd_inst_other"];
10      ->      14      [label="rfo_other"];
10      ->      2       [label="dat"];
10      ->      11      [label="dcb_flush_other"];
10      ->      11      [label="dcb_zero_other"];
10      ->      11      [label="dcb_inval_other"];
11      ->      11      [label="dcb_flush_other"];
11      ->      1       [label="dat"];
```

```
12          ->          12          [label="rd_other"];
12          ->          13          [label="rfo_other"];
12          ->          13          [label="upg_other"];
12          ->          7           [label="dat"];
13          ->          1           [label="dat"];
13          ->          13          [label="rd_other"];
13          ->          13          [label="rd_inst_other"];
13          ->          13          [label="rfo_other"];
13          ->          13          [label="upg_other"];
13          ->          13          [label="upg_stc_other"];
13          ->          13          [label="icb_inval_other"];
13          ->          13          [label="dcb_flush_other"];
13          ->          13          [label="dcb_zero_other"];
13          ->          13          [label="dcb_inval_other"];
14          ->          1           [label="dat"];
14          ->          14          [label="rd_other"];
14          ->          14          [label="rd_inst_other"];
14          ->          14          [label="rfo_other"];
14          ->          14          [label="upg_other"];
14          ->          14          [label="upg_stc_other"];
14          ->          14          [label="icb_inval_other"];
14          ->          14          [label="dcb_flush_other"];
14          ->          14          [label="dcb_zero_other"];
14          ->          14          [label="dcb_inval_other"];
15          ->          15          [label="rfo_other"];
15          ->          15          [label="upg_other"];
15          ->          15          [label="upg_stc_other"];
15          ->          15          [label="dcb_zero_other"];
15          ->          15          [label="rd_other"];
15          ->          15          [label="rd_inst_other"];
15          ->          15          [label="wb_other"];
15          ->          15          [label="dcb_flush_other"];
15          ->          15          [label="dcb_inval_other"];
15          ->          15          [label="icb_inval_other"];
15          ->          17          [label="rd_self"];
15          ->          17          [label="rd_inst_self"];
15          ->          16          [label="dat"];
16          ->          3           [label="rd_self"];
16          ->          16          [label="icb_inval_other"];
17          ->          17          [label="rd_other"];
17          ->          19          [label="rfo_other"];
17          ->          19          [label="upg_other"];
17          ->          19          [label="upg_stc_other"];
17          ->          18          [label="dcb_flush_other"];
17          ->          18          [label="dcb_zero_other"];
17          ->          18          [label="dcb_inval_other"];
17          ->          3           [label="dat"];
17          ->          17          [label="icb_inval_other"];
18          ->          1           [label="dat"];
19          ->          1           [label="dat"];
19          ->          19          [label="rfo_other"];
19          ->          19          [label="upg_other"];
19          ->          19          [label="upg_stc_other"];
19          ->          19          [label="dcb_zero_other"];
```

```
19          ->          19          [label="rd_other"];
19          ->          19          [label="rd_inst_other"];
19          ->          19          [label="wb_other"];
19          ->          19          [label="dcb_flush_other"];
19          ->          19          [label="icb_inval_other"];
19          ->          19          [label="dcb_inval_other"];
20          ->          24          [label="rfo_self"];
21          ->          24          [label="upg_self"];
22          ->          24          [label="upg_self"];
20          ->          24          [label="upg_self"];
21          ->          5           [label="dcb_zero_self"];
22          ->          5           [label="dcb_zero_self"];
20          ->          5           [label="dcb_zero_self"];
20          ->          1           [label="dcb_flush_self"];
20          ->          20          [label="icb_inval_self"];
20          ->          1           [label="dcb_inval_self"];
20          ->          23          [label="dat"];
20          ->          20          [label="rfo_other"];
20          ->          20          [label="upg_other"];
20          ->          20          [label="upg_stc_other"];
20          ->          20          [label="dcb_zero_other"];
20          ->          20          [label="rd_other"];
22          ->          22          [label="rd_other"];
20          ->          20          [label="rd_inst_other"];
20          ->          20          [label="wb_other"];
20          ->          20          [label="dcb_flush_other"];
20          ->          20          [label="icb_inval_other"];
20          ->          20          [label="dcb_inval_other"];
23          ->          5           [label="rfo_self"];
23          ->          5           [label="upg_self"];
24          ->          26          [label="rd_inst_other"];
24          ->          26          [label="rd_other"];
24          ->          25          [label="rfo_other"];
24          ->          25          [label="upg_other"];
24          ->          7           [label="dat"];
25          ->          25          [label="dat"];
25          ->          1           [label="l1_rsp_rfo_other"];
25          ->          25          [label="rfo_other"];
25          ->          25          [label="upg_other"];
25          ->          25          [label="upg_stc_other"];
25          ->          25          [label="dcb_zero_other"];
25          ->          25          [label="rd_other"];
25          ->          25          [label="rd_inst_other"];
25          ->          25          [label="wb_other"];
25          ->          25          [label="dcb_flush_other"];
25          ->          25          [label="icb_inval_other"];
25          ->          25          [label="dcb_inval_other"];
26          ->          26          [label="dat"];
26          ->          26          [label="rd_other"];
26          ->          27          [label="rfo_other"];
26          ->          27          [label="upg_other"];
26          ->          7           [label="l1_rsp_rd_other"];
27          ->          27          [label="dat"];
27          ->          1           [label="l1_rsp_rfo_other"];
```

```
27          ->          1           [label="l1_rsp_rd_other"];
27          ->          27          [label="rfo_other"];
27          ->          27          [label="upg_other"];
27          ->          27          [label="upg_stc_other"];
27          ->          27          [label="dcb_zero_other"];
27          ->          27          [label="rd_other"];
27          ->          27          [label="rd_inst_other"];
27          ->          27          [label="wb_other"];
27          ->          27          [label="dcb_flush_other"];
27          ->          27          [label="icb_inval_other"];
27          ->          27          [label="dcb_inval_other"];
21          ->          5           [label="upg_self"];
28          ->          5           [label="upg_self"];
21          ->          21          [label="rd_other"];
28          ->          28          [label="rd_other"];
21          ->          21          [label="rfo_other"];
28          ->          21          [label="rfo_other"];
21          ->          21          [label="upg_other"];
28          ->          21          [label="upg_other"];
21          ->          21          [label="upg_stc_other"];
28          ->          21          [label="upg_stc_other"];
29          ->          5           [label="upg_stc_self"];
29          ->          30          [label="upg_stc_other"];
29          ->          30          [label="rfo_other"];
29          ->          30          [label="upg_other"];
30          ->          1           [label="upg_stc_self"];
30          ->          30          [label="rfo_other"];
30          ->          30          [label="upg_other"];
30          ->          30          [label="upg_stc_other"];
30          ->          30          [label="dcb_zero_other"];
30          ->          30          [label="rd_other"];
30          ->          30          [label="rd_inst_other"];
30          ->          30          [label="wb_other"];
30          ->          30          [label="dcb_flush_other"];
30          ->          30          [label="icb_inval_other"];
30          ->          30          [label="dcb_inval_other"];
22          ->          5           [label="upg_self"];
31          ->          5           [label="upg_self"];
22          ->          22          [label="rd_other"];
31          ->          31          [label="rd_other"];
22          ->          22          [label="rfo_other"];
31          ->          22          [label="rfo_other"];
22          ->          22          [label="upg_other"];
31          ->          22          [label="upg_other"];
22          ->          22          [label="upg_stc_other"];
31          ->          22          [label="upg_stc_other"];
22          ->          5           [label="dcb_zero_self"];
31          ->          5           [label="dcb_zero_self"];
32          ->          5           [label="upg_stc_self"];
32          ->          32          [label="rd_other"];
32          ->          30          [label="upg_stc_other"];
32          ->          30          [label="upg_other"];
32          ->          30          [label="rfo_other"];
33          ->          33          [label="l1_wb"];
```

```
33          ->          33          [label="dcb_flush_other"];
33          ->          33          [label="dcb_inval_other"];
33          ->          33          [label="icb_inval_other"];
33          ->          37          [label="rd_other"];
33          ->          36          [label="rfo_other"];
33          ->          36          [label="dcb_zero_other"];
33          ->          1           [label="wb_self"];
34          ->          34          [label="l1_wb"];
34          ->          1           [label="wb_self"];
34          ->          35          [label="rfo_other"];
34          ->          34          [label="rd_other"];
34          ->          34          [label="rd_inst_other"];
35          ->          35          [label="l1_wb"];
35          ->          1           [label="wb_self"];
36          ->          1           [label="wb_self"];
37          ->          37          [label="l1_wb"];
37          ->          1           [label="wb_self"];
37          ->          37          [label="icb_inval_self"];
37          ->          1           [label="dcb_flush_self"];
37          ->          1           [label="dcb_inval_self"];
37          ->          37          [label="rd_other"];
37          ->          36          [label="rfo_other"];
37          ->          36          [label="upg_other"];
37          ->          36          [label="upg_stc_other"];
38          ->          7           [label="validate"];
38          ->          7           [label="l1_rsp_rd_other"];
38          ->          7           [label="l1_wb"];
38          ->          7           [label="dcb_flush"];
39          ->          6           [label="validate"];
39          ->          33          [label="l1_rsp_evict"];
39          ->          6           [label="l1_rsp_rfo_other"];
39          ->          6           [label="l1_rsp_upg_other"];
39          ->          6           [label="l1_wb"];
39          ->          6           [label="dcb_flush"];
40          ->          40          [label="l1_wb"];
40          ->          1           [label="e_victimizer_ordered"];
40          ->          40          [label="rd_other"];
40          ->          40          [label="rfo_other"];
41          ->          1           [label="icb_inval_self"];
42          ->          3           [label="icb_inval_self"];
43          ->          5           [label="icb_inval_self"];
44          ->          6           [label="icb_inval_self"];
45          ->          2           [label="icb_inval_self"];
46          ->          7           [label="icb_inval_self"];
}
```