Detecting and Exploiting Causal Relationships in Hardware Shared-Memory Multiprocessors

by

Harold W. Cain III

A dissertation submitted in partial fulfillment of

the requirements for the degree of

Doctor of Philosophy

(Computer Sciences)

at the

UNIVERSITY OF WISCONSIN-MADISON

2004

© Copyright Harold Cain III 2004
All Rights Reserved

Abstract

This thesis focuses on mechanisms that improve inter-processor communication in hardware shared-memory multiprocessors by detecting and exploiting knowledge of the causal relationships among inter-processor reads and writes to shared memory. We present two applications for exploiting causal dependence knowledge: the avoidance of replays in a novel value-based memory ordering mechanism, and the avoidance of coherence misses in an invalidation-based coherence protocol.

Conventional out-of-order processors employ a multiported, fully-associative load queue to guarantee correct memory reference order both within a single thread of execution and across threads in a multiprocessor system. As improvements in process technology and pipelining lead to higher clock frequencies, scaling this complex structure to accommodate a larger number of in-flight loads becomes difficult. The value-based memory ordering mechanism presented here solves the associative load queue scalability problem by completely eliminating the associative load queue. Instead, data dependences and memory consistency constraints are enforced by simply re-executing load instructions in program order prior to retirement. By inferring the existence of causal relationships among processors, the set of loads that must be replayed is filtererd, decreasing the cache bandwidth demands of the load replay mechanism. Consequently, the replay-based mechanism enables a simple, scalable, and energy-efficient FIFO load queue design requiring no associative lookup hardware, while sacrificing only a negligible amount of performance and cache bandwidth.

The overhead of inter-processor communication in shared-memory multiprocessors is a dominant source of processor stalls for many applications. We present a new edge-chasing algorithm for detecting causal relationships in shared memory multiprocessors, and present an implementation of delayed consistency based on this algorithm that is able to avoid coherence misses, allowing a processor to continue reading an invalidated cache block until the processor becomes causally dependent upon a newer version of the block. We have shown that edge-chasing delayed consistency can dramatically improve performance for lock-free list manipulation algorithms that operate on highly-contended data structures, and also improve the performance of some commercial workloads, up to 8% for the applications presented in this thesis.

Acknowledgements

First and foremost, I would like to thank my partner in crime, Heather Swanson, for her support throughout the six-odd years of my quest for a Ph.D. Considering that she married me halfway through this process, it must not have been that bad, but I appreciate that she has tolerated my long hours at work, and my preoccupation with work when I'm home, with minimal complaints. I couldn't have done it without her.

I would like to thank my parents for preparing me for much of what I've encountered since leaving home. I never expected a Kentucky hillbilly like myself to make it this far, so they must have done something right. My dad provided a proof of concept that Cains are indeed capable of earning a Ph.D., and when obstacles seemed insurmountable, their encouragement to "keep pecking at it" helped me realize that it might actually be possible.

Of course, the most significant source of guidance during my research has been my advisor, Mikko Lipasti. The combination of Mikko's technical ability and incredible generosity with his time has been a boon to my development as a researcher. His unfaltering optimism has also been a source of great encouragement. Since I can't thank him enough myself (and Professors' salaries aren't what they ought to be), please thank him on my behalf by buying his new book (*Modern Processor Design: Fundamentals of Superscalar Processors*).

Other faculty and members of my doctoral committee have also helped shape the course of my graduate career. I would especially like to thank David Wood for cementing my interest in computer architecture with his excellent lectur-

ing in CS 752 and Mark Hill for opening my eyes to the inherent difficulties involved with designing multiprocessor systems in CS 757. I've learned a considerable amount from Jim Smith, who has stopped by our lab for a soda on countless occasions, and stayed long enough to impart his experience on many topics, both technical and non-technical. The combined computer architecture faculty have fostered a unique learning environment here at Wisconsin, from tense seminars, prelims and defenses to collegial beer-drinking outings, football playing, and conference trips. You may be skewered by faculty and students during a talk, but can still enjoy a beer with them at the terrace a few minutes later. Afterwards when giving the talk outside the univerity, you're ready for anything that might be thrown your way.

My research has also benefited from my internship with Dr. Ravi Nair at IBM Research. This internship gave me insight into the problems being faced by the design teams building the next generation of computing systems, and opened my eyes to the broad design space of computer systems in general, not just the high-performance general-purpose systems to which I had previously been exposed. I was also introduced to the outstanding research environment at IBM, where I look forward to returning after my graduation.

Gordie Bell, Ilhyun Kim, Kevin Lepak and I were together since the beginning of Mikko's research group, and we have learned a great deal from one another. Although we occasionally managed to get some research done, I'll always remember graduate school for the "character-building" activities that bonded us together: simulator hacking, cluster construction, fighting with Condor, reinstall-

ing all of our systems after getting hacked (twice), and grad student-priced Taco Tuesdays and sometimes Thursdays.

I've received a lot of good advice along the way. Bart Miller taught me to finish what you start, and there cannot be any better advice given to a young graduate student. Dabbling from project to project without ever accomplishing anything is too easy. It is the discipline to finish a project, even if the outcome is not what you hope or expect, that leads to success.

I'm deeply indebted to Ravi Rajwar for the use of his SimpleMP multiprocessor simulator. Ravi was also a source of good advice and many technical discussions in the early stages of my graduate career, for which I am thankful. I would like to thank Pat Bohrer and the rest of the SimOS-PPC group at IBM Research for providing their SimOS-PPC implementation. From SimpleMP and SimOS-PPC, we concocted our own Frankenstein, without which the experiments in this thesis could not have taken place. I would like to thank Kevin Lepak for lifting this heavy burden with me, especially for his development of the multiprocessor verification tools. I would also like to thank Brandon Schwartz (and transitively Jim Smith) for a significant amount of his time designing PHARMsim's address translation unit. Ilhyun Kim was instrumental to the initial transition from our earliest PowerPC-based version of SimpleScalar to SimpleMP, and likewise Gordie Bell to porting PHARMsim to x86/Linux.

My research would not have been possible without the financial support and equipment donations from IBM, Intel, and the NSF. IBM generously donated an 18-processor PowerPC S80 server that was used for debugging PHARMsim

and setting up the workloads used in this thesis. My final year of graduate school was supported by an IBM graduate fellowship, for which I am very thankful. I would like to especially thank Hal Kossman of IBM for his continual support of our research. Intel also contributed significantly to the necessary infrastructure for performing this research, including the workstation and laptop that I've used for the past three years and the motherboards and CPUs in the 200 CPU cluster that serves as our simulation farm. I would also like to especially thank Konrad Lai of Intel for his role in these donations. Prior to the IBM fellowship, my research was supported by National Science Foundation grant numbers CCR-0073440, CCR-0083126, EIA-0103670, CCR-0133437.

This list is by no means complete. There are countless other family members, friends, professors, and staff members who have also helped me along the way. I apologize for blanketing you in here, but I appreciate your support nonetheless. Thank you.

Table of Contents

List of Figures

List of Tables

Chapter 1

Introduction

Computer systems have become an integral part of our daily lives, from the digital alarms that wake us in the morning to the traffic lights that direct our embedded-processor assisted cars home at the end of the day, to their use in all manner of endeavors in-between. Multiprocessor computer systems comprise one part of this infrastructure, providing the computational power that fuels the solution of many important scientific problems, and also enabling the behind-the-scenes infrastructure for our internet-based communications, electronic commerce, and large financial institutions. The performance of these systems is critical to both the smoothness of our routine interactions with them (e.g. the response time of a favorite web page) and the timely solution of many of the world's problems (e.g. an accurate prediction of global climate trends).

The shared-memory programming model is a commonly used interface for communication among multiple independent threads of execution, and multiprocessors that directly implement this interface in hardware constitute a large fraction of all multiprocessor systems in use today. As a consequence of the ever-increasing levels of integration provided by Moore's law [80], the importance of this class of machines will grow; all major computer manufacturers are either currently or will soon be shipping shared-memory multiprocessors manufactured on a single piece of silicon.

Due to the discrepancy between processor clock cycle time and DRAM

access time, all modern computer systems maintain a local copy of recently touched data in a memory structure called a cache. Because most applications exhibit temporal locality, a cache improves performance by allowing the processor to access a recently touched memory location a second time at latency much lower than DRAM. In shared memory multiprocessors, these local copies of memory must be managed to prevent a processor from continuing to read a stale copy of the data after some other processor has written that data.

Invalidation-based coherence protocols have been proven to offer the most bandwidth-efficient mechanism for supporting a coherent shared memory image in a multiprocessor system, and their use has become commonplace. Although decreasing transistor size has improved system performance through decreased clock cycle time, this reduction has unfortunately become a double-edged sword by increasing the relative latency of communication with external sources, regardless of whether those sources are on the other side of the chip or another chip, circuit board, or computer altogether. Consequently, modern processors spend a significant fraction of their time sitting idle, waiting for a memory reference that could not be serviced by its local cache hierarchy and instead must be transmitted by a more distant source. In invalidation-based coherence protocols, when one processor is writing a particular memory location, that block is removed from the caches of other processors. Should those processors subsequently access the location, their access will incur cache misses and most likely result in processor stalls. Consequently, cache misses due to inter-processor communication significantly degrade performance for many parallel applications.

Another effect of decreasing clock cycle time is that the distance that can be traveled by a signal within this period also shrinks. Due to this shrinking distance, many components of a microarchitecture must either shrink accordingly or must be redesigned in such a way that allows pipelined access. The microarchitectural structures that enforce the proper ordering of interprocessor reads and writes are complicated by this desire for low latency and the need for large capacity.

The objective of the research presented in this thesis is to enhance the performance of shared-memory multiprocessor systems by solving these problems associated with inter-processor communication. Our solutions are based on the observation that non-causally related inter-processor communication should not degrade performance or increase the complexity of a design. By identifying instances of communication between two processors that are not causally related and filtering the performance degrading effects of this communication, we can improve system performance.

Given a system composed of multiple processes where each process performs a sequence of events including inter-process send and receive operations, Lamport defined the causality relation which specifies the necessary order of events in the system [57]. The causality relation states that the order in which events at one process become observable to other processes should reflect the sequence in which they occur within each process. Informally, in a shared-memory multiprocessor, when one processor reads a memory location that was previously written by another processor, or when one processor overwrites a memory location that has been previously read or written by other processors, that processor

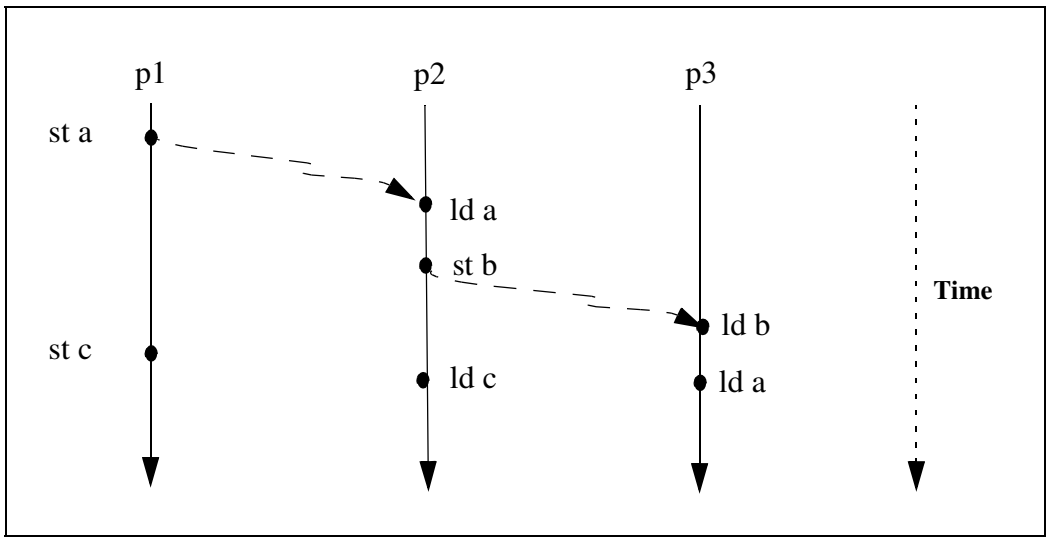

FIGURE 1-1. Causal dependencies in a shared-memory multiprocessor system.

becomes causally dependent upon those prior read and write operations performed by other processors. Such causal dependences are transitive in nature, as illustrated in Figure 1-1 in the context of a system consisting of three processors (*p1*, *p2*, *p3*) and three memory locations (*a*, *b*, *c*). Processor *p1* initially performs a store to memory location *a*, and *p2* subsequently reads the newly written value. Processor *p2* is now causally dependent upon *p1*'s store to *a* and those instructions on which *p1*'s store to *a* causally depend. Processor *p2* subsequently writes location *b*, which is then read by *p3*. When *p3* performs a load to location *a*, it is already causally dependent upon *p1*'s store to *a* transitively through the memory location *b*, and must therefore read the value written by *p1*. Two events are said to be concurrent if neither event is causally dependent upon the other. When *p2* executes its *ld c*, it is not already causally dependent upon *p1*'s store to *c*. Therefore, *p2*'s load to *c* may correctly return either the value written by *p1* or the value that existed at *c* prior to *p1*'s write. It is this type of ambiguity that the techniques presented in this thesis exploit. We provide a more formal definition of these causal relationships in

Chapter 2, adapting this simplified description to the more complex set of memory consistency rules dictated by modern computer architectures.

This thesis focuses on the detection of such causal relationships, and the use of this knowledge to optimize the performance of hardware shared-memory multiprocessors. We present two applications for exploiting causal dependence knowledge: the avoidance of replays in a novel value-based memory ordering mechanism, and the avoidance of coherence misses in an invalidation-based coherence protocol. In the next two sections, we present an overview of these applications, followed by a summary of the contributions made by this thesis and an outline of its contents.

1.1 Value-Based Memory Ordering

Conventional out-of-order processors employ a multiported, fully-associative load queue to guarantee correct memory reference order both within a single thread of execution and across threads in a multiprocessor system. As improvements in process technology and pipelining lead to higher clock frequencies, scaling this complex structure to accommodate a larger number of in-flight loads becomes difficult if not impossible. Furthermore, each access to this structure consumes excessive amounts of energy. In the first part of this thesis, we present a novel memory ordering mechanism that solves the associative load queue scalability problem by completely eliminating the associative load queue. Instead, data dependences and memory consistency constraints are enforced by simply re-executing load instructions in program order prior to retirement. Should a load be

incorrectly reordered with respect to a prior store, or with respect to another processor's invalidation message, the re-execution mechanism will detect the error and force the problematic load to be squashed.

Naively, all loads should be re-executed to guarantee a correct execution. Unfortunately there are two primary costs associated with replaying loads, which we would like to avoid: 1) load replay can become a performance bottleneck given insufficient cache bandwidth for replays or due to the additional resource occupancy, and 2) each replayed load causes an extra cache access and word-sized compare operation, consuming energy.

It is in this context that obtaining knowledge of the causal relationships among processors can be advantageous. If one can detect the instances in which a load necessarily received a correct value when it originally accessed the cache, the costs associated with replay can be avoided. We describe and evaluate a set of heuristics that can be used to perform this detection for both uniprocessor read-after-write hazards and multiprocessor ordering violations. These heuristics infer the potential existence of causal dependences based on locally observable events, and we find that this inference is very effective at reducing the number of loads that must be replayed. Consequently, the replay-based mechanism enables a simple, scalable, and energy-efficient FIFO load queue design requiring no associative lookup hardware, while sacrificing only a negligible amount of performance and cache bandwidth.

1.2 Edge-Chasing Delayed Consistency

In shared memory multiprocessors utilizing invalidation-based coherence protocols, cache misses caused by inter-processor communication are a dominant source of processor stall cycles for many applications, particularly commercial workloads. In the second part of this thesis, we explore a novel delayed consistency implementation called edge-chasing delayed consistency that attempts to mitigate the performance degradation caused by this class of misses. Edge-chasing delayed consistency allows a processor to continue to non-speculatively read a cache block after it has been invalidated from its local cache hierarchy. Using the example from Figure 1-1, it is technically correct for *p2*'s load to location *c* to return either the value written by *p1* or the value that existed before *p1*'s write, because the operations are concurrent. If *p2*'s load can return either value, then there is no reason to penalize this load by forcing it to wait while obtaining the newest value of *c*.

The edge-chasing consistency protocol identifies those cache blocks that can safely be read after being invalidated by tracking the operations that are causally dependent upon that cache block. While the replay reduction heuristics used in the first part of this thesis infer causal dependences based on locally observable events, the edge-chasing delayed consistency protocol explicitly tracks the causal dependences among processors, forcing a processor to obtain a new copy of a block only after the processor becomes causally dependent upon the write that invalidated the block.

Because the expiration of a cache block's useful lifetime is based on cau-

sality, this expiration is delayed until it is dictated by the memory consistency model, longer than it would have been delayed by prior delayed consistency implementations. In our evaluation of edge-chasing delayed consistency, we find that the performance of some applications can be improved by this implementation technique. Of four commercial workloads studied, edge chasing delayed consistency improves the performance of two, TPC-H and SPECweb99, by 8% and 4% respectively. We find edge-chasing delayed consistency has little effect on the performance of the scientific applications studied here.

1.3 Thesis Contributions

- **Elimination of associative search logic from the load queue via value-based replay**: We demonstrate a value-based replay mechanism for enforcing uniprocessor and multiprocessor ordering constraints that eliminates the need for associative lookups in the load queue, creating a more scalable memory ordering implementation for large window out-of-order microprocessors.
- **Replay-reduction heuristics:** We introduce several novel heuristics that reduce the cache bandwidth required by the load-replay mechanism to a negligible amount.
- **Consistency model checking:** we define the constraints for implementing a back-end memory consistency checker. These constraints are also useful in the domain of other checking mechanisms such as DIVA [10]. Recent work has exposed a subtle interaction between memory consistency and

value prediction that results in consistency violations under forms of weak consistency [71]. Our value-based replay implementation may be used to detect such errors.

- **Edge-chasing delayed consistency:** we describe a novel implementation of delayed consistency that extends the lifetime of cache blocks beyond the time that they are invalidated, remaining useful until the processor becomes causally dependent upon a newer version of the block. This implementation is based on a novel edge-chasing algorithm for tracking causality that may prove more generally applicable than the delayed consistency mechanism discussed here. We show that the edge-chasing delayed consistency protocol can improve the performance of some applications significantly.
- **Constraint-graph extensions:** We describe extensions to the constraint graph model for reasoning about memory models other than sequential consistency. In addition to the implementation techniques discussed here, these extensions are broadly applicable to the design and verification of multiprocessor systems that employ a relaxed memory model.

1.4 Thesis Organization

This thesis is organized as follows. In Chapter 2, we present the formal framework used for our discussion of causality, based on the constraint graph representation originally defined by Landin, Hagersten, and Haridi [59]. Before presenting the design and evaluation of value-based memory ordering and edge-

chasing delayed consistency, Chapter 3 includes a description of the methodology used for the experimental sections of the thesis, including PHARMsim, the PowerPC-based full-system timing simulator developed for this work. In Chapter 4, we describe the value-based memory ordering mechanism, followed by a performance evaluation of the mechanism relative to an aggressive conventional load queue design in Chapter 5. We present the design and implementation of the edge-chasing delayed consistency protocol in Chapter 6, followed by an experimental evaluation of the protocol in chapter 7. A discussion of prior work and its relationship to the different parts of this thesis is separated by topic at the end of chapters 2, 4, and 6.

Chapter 2

Formal Representations of Shared Memory

Since the invention of shared-memory multiprocessor computers, there has been a plethora of research that formally defines the behavior of the shared-memory interface. These definitions have used a wide range of specification languages, for example I/O automata [42], axiomatic specifications [27][100], set and function-based specifications [30], graph based representations [1][29][39][59], and temporal logic [58]. In this thesis, we utilize one of these formal representations, the constraint graph model originally proposed by Landin, Hagersten, and Haridi [59]. Each of these proposals is essentially a representation of the same shared memory interface using a different formalism (most models have been defined with respect to a sequentially consistent memory, with some including extensions for other memory models). The choice of a formalism should depend on the purpose for which it will be used. For example, if one plans to verify the correctness of a coherence protocol, a detailed specification of machine behavior is appropriate, such as a specification written in TLA+. In this thesis, we use a formal model to reason about the correctness of multithreaded executions, and we consequently choose the constraint graph for its simplicity. The directed acyclic graph is a commonly understood abstraction, which will hopefully prevent our discussion from being obfuscated by overly formal language.

This chapter is organized as follows: in Section 2.1 we formally describe the constraint graph model previously defined by Landin et al., and describe our

extensions that allow it to be used when reasoning about processor consistent and weakly ordered memory models. In Section 2.2 we informally discuss how the constraint graph may be used to reason about the correctness of multithreaded executions on shared-memory multiprocessors through a set of examples, and explain how our augmented definition differs from the original constraint graph definitions. We conclude in Section 2.3 with a more thorough discussion of other formal shared memory representations and related work.

2.1 Constraint Graph Model and Extensions

An execution of a multithreaded program can be represented in terms of a directed graph called a *constraint graph*, consisting of a set of vertices $V = \{ v_1, v_2, ... v_n \}$ and edges $E = \{ e_1, e_2, ..., e_m \}$. Vertices in the graph represent dynamic instances of memory operations[1], and edges represent the transitive ordering relationships among these operations. This representation was originally defined by Landin et al. (who used the term *access graph*) for reasoning about the correctness of request reorderings in sequentially consistent shared-memory multiprocessor interconnection networks. Condon and Hu subsequently re-invented the model for use when verifying that a machine is sequentially consistent [29]. The key idea behind the constraint graph that makes it a powerful tool for reasoning about multithreaded executions is that it can be used to test the correctness of an execution by simply testing the graph for a cycle. The presence of a cycle indicates that the

1. In RISC architectures, such operations are typically ordinary load and store instructions, however in architectures with memory instructions that perform multiple non-atomic memory operations, such nodes correspond to the individual non-atomic memory operations.

execution violates the consistency model because the observed order of operations cannot be placed in an order that agrees with the order dictated by the memory model.

The constraint graph was originally defined solely for sequential consistency, but we would also like to use it to reason about more relaxed memory consistency models. In this section, we define the constraint graph model with extensions that allow one to reason about the necessary ordering among operations, whether the system is sequentially consistent or uses a relaxed memory model. This definition differs from the previous definitions by adding or removing certain edge types from the graph, depending on the consistency model, as described below.

The following set of relations describes the edge types in our augmented constraint graph representation. These ordering relations are divided into *single-thread ordering relations* and *dynamic ordering relations*.

2.1.1 Single-thread Ordering Relations

The single-thread ordering relations place an order on the operations that are executed by a single processor. These relations are derived from the sequence of operations executed by a single thread, and they exist regardless of interactions with other threads. Depending on the consistency model, some of these orders may or may not be enforced. The following 15 single-thread ordering relations represent the ordering constraints enforced or relaxed by all current commercial architectures. These single-thread relations can be divided into four groups: program order relations, same address relations, data dependence relations, and memory

1. Program order-based relations ($\overrightarrow{po}$, $\overrightarrow{rdrd}$, $\overrightarrow{wrwr}$, $\overrightarrow{rdwr}$, $\overrightarrow{wrrd}$): these five relations order operations in the order specified by each processor's program. An operation *i* precedes an operation *j* in *program order* ($\overrightarrow{po}$) if both operations are executed by the same processor *p*, and operation *i* precedes operation *j* in the order specified by *p*'s program. Program order can be further divided into r*ead-to-read order* ($\overrightarrow{rdrd}$), *write-to-write order* ($\overrightarrow{wrwr}$), *read-to-write order* ($\overrightarrow{rdwr}$) and *write-to-read order* ($\overrightarrow{wrrd}$), depending on whether *i* and *j* are reads or writes.

2. Same address ordering relations ($\overrightarrow{rdrd-sa}$, $\overrightarrow{wrwr-sa}$, $\overrightarrow{rdwr-sa}$, $\overrightarrow{wrrd-sa}$): these four relations define an ordering between operations that reference the same memory location. For example, an operation *i* precedes an operation *j* in *same-address read-to-read order* ($\overrightarrow{rdrd-sa}$) if both operations read from the same memory location, and $i \overrightarrow{po} j$. These relations specify the constraints on operations that access the same memory location in consistency models that do not already enforce the program order-based relations.

3. Data dependence ordering relations ($\overrightarrow{rdrd-dep}$, $\overrightarrow{rdwr-dep}$): these two relations define an ordering between memory operations whose addresses depend on a prior memory read. For example, an operation *i* precedes an operation *j* in *data dependent read-to-read order* ($\overrightarrow{rdrd-dep}$) if both operations read from memory and the address computation used to perform *j* is data dependent on the value returned by *i*, and $i \overrightarrow{po} j$. An operation *i* precedes an operation *j* in *data dependent*

read-to-write order ($\overrightarrow{rdwr-dep}$) if operation *i* reads from memory and operation *j* writes to memory, and the address computation used to perform *j* is data dependent on the value returned by *i* or the value written by *j* is data dependent on the value returned by *i*, and *i* $\overrightarrow{po}$ *j*.

4. Memory ordering instruction (MOI) relations ($\overrightarrow{rdmoi}$, $\overrightarrow{wrmoi}$, $\overrightarrow{moird}$, $\overrightarrow{moiwr}$): these four relations define an ordering between the memory ordering instructions defined by an architecture (e.g. PowerPC *sync*, IA-64 *st.rel*, etc.), and read/write operations. The MOI relations required by an architecture depend on the semantics of the memory ordering instruction. For example, the PowerPC *sync* instruction enforces a total order from all memory operations that precede the instruction to all memory operations that follow it. Consequently, all of MOI relations ($\overrightarrow{rdmoi}$, $\overrightarrow{wrmoi}$, $\overrightarrow{moird}$, $\overrightarrow{moiwr}$) are used to specify its semantics. On the other hand, the Alpha's write memory barrier (`wmb`) instruction only forces an ordering among writes, so only the $\overrightarrow{wrmoi}$ and $\overrightarrow{moiwr}$ relations are used to order memory operations with respect to the *wmb*.

2.1.2 Dynamic Ordering Relations

The dynamic ordering relations ($\overrightarrow{dyn-raw}$, $\overrightarrow{dyn-waw}$, $\overrightarrow{dyn-war}$) reflect the orders among operations that are dynamically observed during a particular program execution. Each of these relations is address dependent, meaning that two operations will only be ordered by dynamic ordering relations if they access the same memory location. The dynamic ordering relations are consistency model independent; each of these ordering relations is present regardless of the memory

consistency model. Unlike the single-thread ordering relations, the dynamic ordering relations place an order on memory operations executed by different processors, in addition to operations executed by the same processor.

- **Dynamic read-after-write order** ($\overrightarrow{dyn-raw}$): An operation *i* precedes an operation *j* in dynamic read-after-write order if operation *i* writes the memory location at address *a* and operation *j* reads the memory location at address *a*, and *j* reads the value written by *i* before it is overwritten.
- **Dynamic write-after-write order** ($\overrightarrow{dyn-waw}$): An operation *i* precedes an operation *j* in dynamic write-after-write order if operations *i* and *j* write the same memory location, and *j* overwrites the value written by *i*.
- **Dynamic write-after-read order** ($\overrightarrow{dyn-war}$): An operation *i* precedes an operation *j* in dynamic write-after-read order if operation *i* reads the memory location at address *a* and operation *j* writes the memory location at address *a*, and *j* overwrites the value read by *i*.

2.2 Discussion

This constraint graph definition differs from prior definitions by adding single-thread ordering constraints other than $\overrightarrow{po}$. In the absence of these other single-thread constraints, this representation is equivalent to the previous definitions. It should be clear that there is overlap between many of the single-threaded ordering relations. For example, any two operations that are ordered by $\overrightarrow{rdwr}$ are also ordered by $\overrightarrow{po}$. Program order is the union of each of the other orders; each is a subset of program order. The relationship between the single-threaded orders is

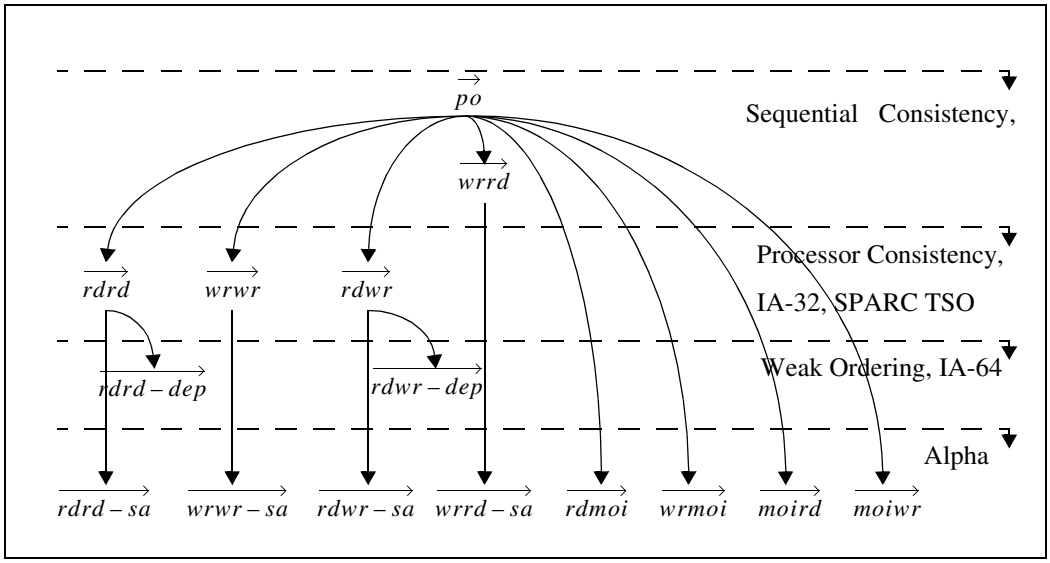

FIGURE 2-1. Relationship between single-thread ordering constraints.

illustrated in Figure 2-1. This figure also separates the single-threaded orders by consistency model. Below each labeled dotted line are the orders that must be enforced to correctly implement a specific consistency model. For example, in order for a system to implement processor consistency, it is not necessary to enforce $\overrightarrow{po}$ or $\overrightarrow{wrrd}$, however the remaining orders must be enforced.

2.2.1 Definition of causal dependence

Throughout this thesis, we use the term causally dependent to refer to the relationship of operations in the constraint graph. An operation i_1 is *causally dependent* upon an operation i_2 if there exists a directed path from i_2 to i_1 in the constraint graph. We also use the statement that a processor p is causally dependent upon an operation i_1, meaning that some prior operation executed by p was causally dependent upon i_1, therefore any subsequent operation executed by p will also be causally dependent on i_1. We also use the terms *upstream* and *downstream* to specify the relationship of operations in the constraint graph. An operation i_1 is

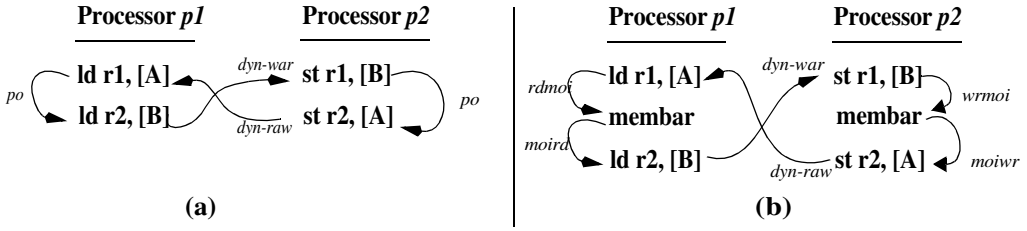

FIGURE 2-2. Constraint graph examples. (a) Sequentially inconsistent execution (b) Equivalent weakly ordered inconsistent execution

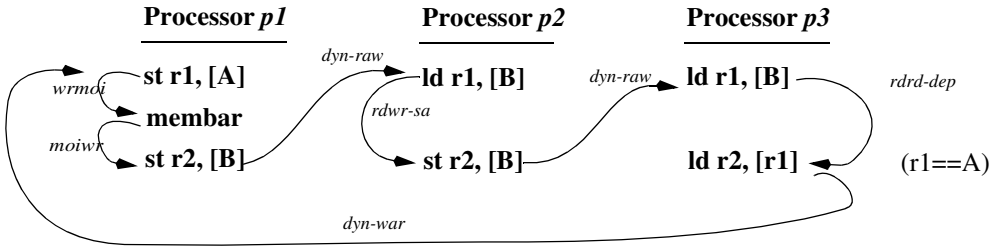

FIGURE 2-3. Dependence order/Same-Address order in a weakly ordered system.

upstream from an operation i_2 if there is a directed path from i_1 to i_2. If i_1 is upstream from i_2, then i_2 is downstream from i_1.

2.2.2 Examples

Figure 2-2, Figure 2-3, and Figure 2-4 illustrate how the constraint graph is used to reason about memory consistency models. In each of these examples, redundant edges have been removed for clarity. Figure 2-2 (a) shows an execution that violates sequential consistency, because processor *p1*'s second load instruction should return the value of memory location B written by *p2*, instead of the original value of B. As shown in the figure, the constraint graph corresponding to this execution contains a cycle, indicating that it is erroneous. The example in Figure 2-2 (b) is equivalent to the example in Figure 2-2 (a), only it is written for a weakly ordered memory model. In this erroneous execution, a cycle exists in the

constraint graph which spans the additional memory barrier relative edges. If the memory barrier were not present in either *p1* or *p2*'s instruction stream, the pictured interleaving of loads and stores would be legal. However, because both of the memory barriers are present, if *p1* observes *p2*'s write to address A, then *p1* must also observe *p2*'s write to address B.

Figure 2-3 illustrates a slightly more complicated scenario, in which processor *p3*'s second load instruction erroneously reads the original value of memory location A, rather than the value written by *p1*. Due to the memory barrier instruction at *p1*, the same-address writes at *p2*, and the data dependence at *p3*, the load operation should be ordered after the store operation. If any of these edges were not present (e.g. if *p3*'s operations were not data dependent), then the resulting execution would be correct. These examples illustrate that if there is a cycle in the constraint graph, then the execution is erroneous. However, by the definition of the constraint graph and consistency model, the converse of this is also true: if the constraint graph corresponding to an execution is acyclic, then that execution is correct with respect to the consistency model.

Figure 2-4 shows a constraint graph for another PowerPC execution, corresponding to the executed instruction trace shown in the left side of the figure. In this example, processor *p1* executes a series of instructions between two sync operations. The first three store instructions all write to memory location A, creating a series of $\overrightarrow{wrwr-sa}$ edges connecting them. The next two instructions are a load followed by a data dependent load instruction, which are connected by a $\overrightarrow{rdrd-dep}$ edge in the constraint graph. The following three instructions are a store

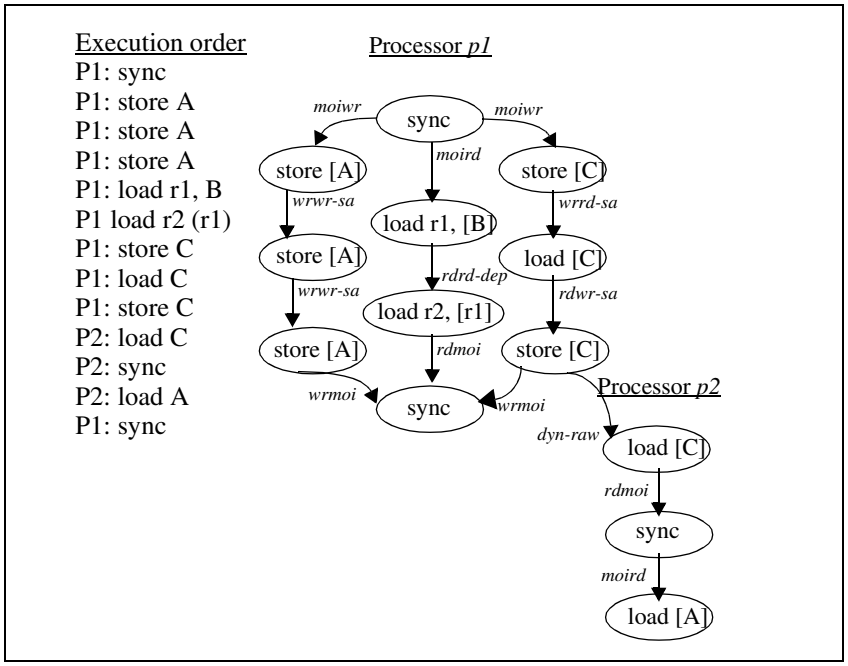

FIGURE 2-4. Weakly ordered constraint graph. Register specifiers omitted when irrelevant

to location C, followed by a load to C, followed by another store to C, which are all ordered with respect to one another via $\overrightarrow{wrrd-sa}$ or $\overrightarrow{rdwr-sa}$ edges because they each read or write the same location.

The resulting constraint graph contains three separate independent strands of execution that are not ordered with respect to one another. Within each strand, the operations must appear to execute in a certain order, but across strands, no ordering guarantees are made. Each strand must appear to execute after the first sync operation and before the second sync operation. When processor *p2* executes its load instruction to C, receiving the data written by *p1*, the load instruction is ordered after the entire right-most strand of *p1*'s execution, but is not ordered with respect to the remainder of *p1*'s operations. Consequently, the load A executed by *p2* does not necessarily need to return any of the values stored by *p1*. It could correctly return any of the values, or the value that existed prior to any of *p1*'s store

operations. Note that if there were a WAR edge from *p2*'s load A to any of *p1*'s stores, the constraint graph would remain acyclic. This example illustrates the additional ordering flexibility garnered by a relaxed memory model.

2.2.3 Write atomicity

Any formal model treads a fine line between including too many details thus making its use unwieldy, and not sufficiently modeling important aspects of the system. The constraint graph model described above neglects modeling non-atomic writes (present in models that allow unordered stores to forward data to subsequent loads, or models that relax the global atomicity of writes). For example, Figure 2-5 illustrates a variation of Dekker's algorithm in which the forwarding of store data from a write buffer can be inferred by the program's outcome. If the value obtained by each processor's first load is forwarded from that processor's store before the store has been globally ordered, and the second load reads the value that existed before each of the stores, then it will be visible to the programmer that the store operation was performed non-atomically. Figure 2-6 illustrates another example where the non-atomicity of writes is visible to the programmer, in this case, because *p1*'s store becomes visible to processor *p2* before processor *p3*.

Because the augmented constraint graph representation above does not include support for the modeling of non-atomic writes, when a constraint graph is created for each of these executions (shown in the figures), a cycle forms indicating that the execution is incorrect. However, in some memory models, these particular executions are correct (e.g., TSO allows the execution in Figure 2-5;

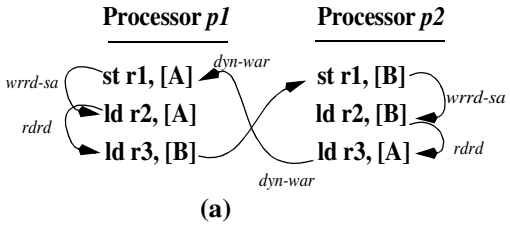

FIGURE 2-5. Detectable instance of store-to-load forwarding from write buffer.

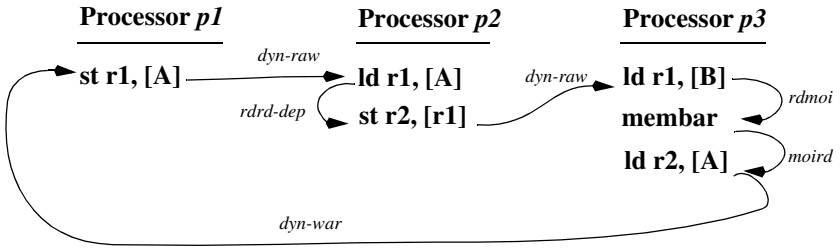

FIGURE 2-6. Detectable instance of non-atomic write.

PowerPC allows the execution in Figure 2-6). Thus, if one wishes to model executions of systems that break write-atomicity, then the constraint graph corresponding to those executions may sometimes indicate that the execution is incorrect when it is not.

When evaluating replay-based memory ordering, we find that there are so few replays performed to enforce sequential consistency that we do not explore weaker models. Consequently, this application is not affected by the omission of non-atomic write modeling. Our evaluation of edge-chasing delayed consistency may be affected by this omission, because edge-chasing delayed consistency may benefit from the relaxation of write atomicity. The lack of modeling for non-atomic writes affects the implementation by creating a cycle when one would otherwise not exist, causing a loss in performance by forcing a processor to unnecessarily stop using a stale cache line. However, exploiting the non-atomicity

property adds a significant amount of complexity to the protocol, which we do not believe would be justified. Prior representations separately proposed by Collier, Adve, and Gharachorloo do include the modeling of non-atomic writes [1][27][39]. Should future implementations of edge-chasing delayed consistency benefit from a more exact enforcement of the necessary conditions for using stale cache lines, one of these more detailed representations can be used instead of our representation.

2.3 Related Work

This work is based on the original constraint graph definition presented by Landin, Hagersten, and Haridi, who used it to reason about the correctness of request and response reordering in sequentially consistent interconnection networks [59]. Prior to Landin et al.'s formalization, a similar model was used by Shasha and Snir to determine the necessary conditions for the insertion of memory barriers by parallelizing compilers to ensure sequentially consistent executions [98]. The constraint graph has since been used to automatically verify sequentially consistent systems [29][90] and reason about the correctness of value prediction in multiprocessor systems [71]. It has also recently been used to measure the differences in the limits of parallelism across consistency models in several multithreaded applications [20], and check the correctness of an implementation of the TSO memory model [44].

Although previous work has defined constraint graphs for sequentially consistent systems, the majority of multiprocessors shipping today employ a

relaxed memory model [49][50][73][101][110]. The representation presented here augments the constraint graph representation for use in reasoning about processor consistent and weakly ordered systems. These extensions are more broadly applicable to reasoning about relaxed memory consistency models, whether it be for the purpose of verification or for the performance analysis discussed here.

There has been a plethora of work on other formalisms for reasoning about shared-memory consistency models. Dubois et al. established the theoretical correctness of delaying data writes until the next synchronization operation [34], and introduced the concept of the memory operations being "performed" locally, performed with respect to other processors in the system, and performed globally. Collier presented a formal framework that divided memory operations into several sub-operations in order to model the non-atomicity of memory operations, and formulated a series of rules regarding the correct observable order of sub-operations allowable in different consistency models [27]. Adve incorporated the best features of Landin et al.'s model (acyclic graph) and Collier's framework (multiple write events) into a powerful unified model, which was used to perform a design space exploration of novel consistency models [1] (specifically Chapter 7). Chapter 4 from Gharachorloo's thesis describes a framework for specifying memory consistency models that improves upon previous representations through the addition of a memory operation "initiation" event, which can be used to model early store-to-load forwarding from a processor's write buffer. The formal representation described in Section 2.1 is most similar to Adve's representation, although in favor of simplicity it lacks the detailed modeling of non-atomic writes.

Chapter 3

Experimental Methodology

In this chapter we describe the methodology used for performance evaluation throughout the thesis. We begin with a discussion of performance evaluation methodologies in general, motivating our desire to create a simulation environment capable of simulating shared-memory multiprocessors executing non-trivial real-world applications. Section 3.1 also describes the machine model implemented by PHARMsim, detailing the modifications to the original SimpleScalar and SimpleMP simulators necessary for supporting modern microarchitectures, system-level simulation, and the PowerPC architecture. We present an overview of the benchmark programs used for performance evaluation in Section 3.2, followed in Section 3.3 by a discussion of the non-determinism problem and its effects on the data presented in this thesis.

3.1 Simulation Environment

All data presented in this thesis was collected using PHARMsim, a PowerPC-based execution-driven full-system timing simulator. PHARMsim was developed by members of the PHARM research group but inherits much of its source code from two pre-existing simulators: the PowerPC port [52] of the SimOS full-system simulator originally developed at Stanford [96], and SimpleMP, a version of the SimpleScalar 3.0 simulator [17] modified by Ravi Rajwar to support multiprocessor systems [91]. In this section, we describe our motiva-

tions for the creation of PHARMsim, present a summary of the features PHARMsim inherits from other simulators, and provide an overview of the enhancements made to the SimpleMP timing simulator to functionally support the PowerPC architecture and full-system simulation, and to support faithful performance modeling of modern microarchitectures.

3.1.1 Why execution-driven simulation?

Execution-driven simulation has become a ubiquitous method of performance estimation in the computer architecture research community, due largely to its flexibility. An execution-driven simulator is a piece of software that models a computer system's functionality, its accuracy being limited only by programming effort, simulation time, and virtual memory size. Arbitrary programs may be executed on the simulator, which functionally executes the semantics of each instruction in the program's dynamic instruction stream, and faithfully calculates the time required to execute the program based on the specifications of the modeled system. Once a simulator has been written for a desired system, it is fairly simple to modify that simulator to evaluate new experimental features.

Alternative approaches to performance analysis include the use of analytic models and trace-driven simulation. Analytic modeling techniques include methods such as bounding calculations (e.g. peak floating point performance = number of floating point units x megahertz), queuing models, and Markov chains. Depending on the level of detail included, such techniques can provide performance estimates ranging from coarse "back-of-the-envelope" calculations to minutely detailed predictions. Unfortunately, detailed analytic microarchitectural processor

models require the collection of much empirical data for their validation and use (in the form of application-specific statistics such as branch misprediction rates or cache miss rates). If this data is collected using simulation, the utility of the analytic model is diminished to that of a validation mechanism, because a simulator already exists that can provide performance estimates. Consequently, their use is often relegated to high-level design-space exploration during the early phases of a design, and simulation is subsequently used to analyze more minute performance trade-offs.

Trace-driven simulation, in which a fixed record of executed instructions or memory references is fed to a simulator that models the timing of a system executing that trace, offers many of the same advantages as execution-driven simulation. Practically any system may be modeled, but the ability to study certain optimizations is limited by the information provided by the trace. For example, if one were attempting to evaluate value-prediction [67] using trace-based simulation, the trace must include values. It is also difficult to precisely model wrong-path speculative execution using trace-based simulation because it is unknown at trace-generation time where a wrong-path instruction stream might venture. (This problem may be solved through the use of a trace-generator running in parallel with the timing simulator, whose instruction stream can be redirected to temporarily follow incorrect paths [37][72][82].)

Attempting to compare multiprocessor simulation results using trace-based simulation is problematic because the constraints placed on the execution by a fixed trace can affect the outcome of the comparison, and the magnitude of this

effect is dependent upon the type of optimization being evaluated. When evaluating an optimization that changes the timing of a program's inter-thread dependences, the optimization's impact may be over or under-estimated because the optimized execution is constrained by a fixed trace. For example, suppose a designer is evaluating an optimization that decreases the latency of inter-processor cache block transfers in a coherence protocol for those blocks containing lock variables. In an actual system, such an optimization may decrease the number of spin iterations executed by processors waiting to acquire the lock. If one were to evaluate this optimization using trace-driven simulation, the spinning processors would continue to execute the same number of spin instructions, even though the lock release should have been observed earlier, and these additional spin instructions would offset any gains obtained through the optimization. Using a fixed trace artificially forces a timing simulator to follow that trace, whereas in a real system the lock transfer optimization would have caused fewer spin loop iterations in the application's execution. Execution driven simulation overcomes this problem when modeling multiprocessor systems, because timing-dependent interactions with other processors and interrupts occur as they would in a real system.

Unfortunately, execution-driven simulation's ability to accurately model a system's non-deterministic behavior can become a liability when attempting to compare two machine configurations if the amount of non-determinism is greater than the expected performance difference between the two machines. We discuss methods for making accurate performance comparisons in light of non-deterministic workloads in Section 3.3.

Due to the advantages described above, execution-driven simulation is the sole simulation methodology that can provide accurate performance estimates for multiprocessor systems. However, not all execution-driven simulators are built alike. One technique that is sometimes used to decrease simulator complexity is to functionally execute instructions in program order and only model the timing of out-of-order instruction execution. Another technique functionally executes instructions using a flat memory hierarchy, and only simulates the timing of a cache hierarchy. Neither of these approaches provide credible performance estimates for the mechanisms studied in this thesis, which depend heavily on the observed execution order and the potential existence of multiple concurrent versions of a single memory location. For example, the performance of the value-based replay scheme described in Chapter 4 depends heavily on the interaction of instruction reordering by a processor's out-of-order instruction window with external invalidations. If the simulator functionally executed instructions in program order, the value returned by load instructions might differ from a simulator that performs functional execution out-of-order, due to a different resolution of a race with an external invalidation. The edge-chasing delayed consistency mechanism described in Chapter 6 is heavily dependent upon the existence of more than one version of a memory location; it is not possible to model the timing of stale value usage without actually using stale values. Consequently, for the techniques studied in this thesis, only a true value-passing "execute during execute" simulator will suffice, which we have created in PHARMsim.

3.1.2 Why full-system simulation?

Since the SimOS project originally demonstrated the feasibility of full system simulation [96], the technology has widely proliferated ([5][11][13][56][52][69]) as an alternative to the user-level/system call proxying simulators typically used for computer architecture research [17][86]. Full-system simulation offers two major advantages over these simulators:

- **Targetability**: the ability to execute any program, regardless of the arcane system calls that program might use, such as large software packages like databases and web servers.
- **Increased Accuracy**: the inclusion of system devices such as disks and network adapters and the effects of those devices on other parts of the system (e.g. cache invalidations due to DMA traffic). Also included is support for supervisor-level instructions within the processor module, resulting in the ability to model the timing of all instructions in a workload, not just user-level instructions. This support is particularly important for I/O intensive applications such as commercial workloads which spend a significant fraction of their time executing system-level code [74][95]. Our research has shown that the inclusion of system-level instructions can also cause a significant performance impact on non-I/O intensive workloads such as the SPEC CPU benchmarks, which were traditionally thought to be unaffected by operating system activity [19].

3.1.3 SimpleScalar 3.0, SimpleMP, and SimOS-PPC functionality

As mentioned earlier, PHARMsim is based on three pre-existing simulators, and consequently inherits the functionality of each of those simulators.

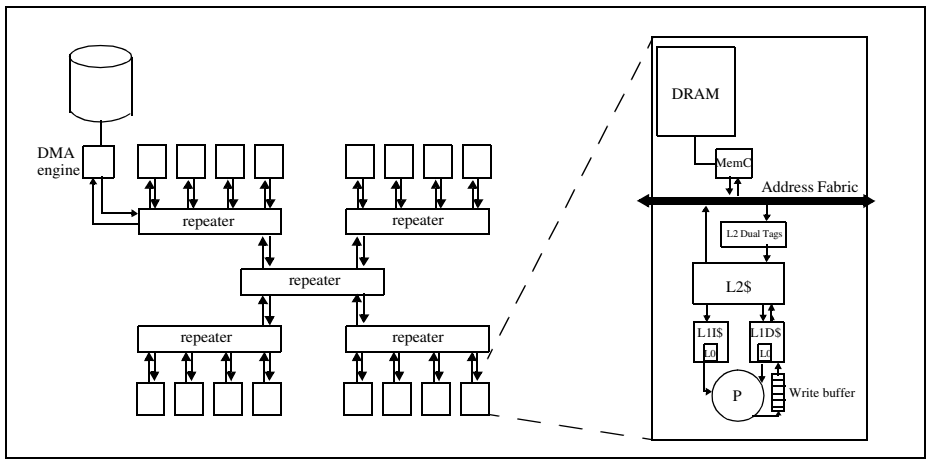

FIGURE 3-1. 16-processor broadcast snooping memory system. (Switched data network not shown.)

PHARMsim relies on SimpleMP for detailed processor and memory system timing modeling. The SimpleScalar 3.0 code on which SimpleMP is based models an out-of-order superscalar microprocessor implemented as a five-stage pipeline, supporting speculative execution and a variety of branch predictors.

SimpleMP is a multiprocessor version of SimpleScalar, augmenting the simulator with two detailed coherent memory systems: a snooping protocol based on the Sun Gigaplane-XB [24], illustrated in Figure 3-1, and a NUMA directory protocol based on the SGI Origin [60], both using inclusive two-level caches. These memory systems model latency and bandwidth at all stages. While SimpleScalar uses a flat image of memory and simulates the timing of a cache hierarchy, SimpleMP's processor reads data contained in the caches as would occur in an actual system; consequently, the timing and values of reads to shared data reflect those which would occur in a real system. Similarly, SimpleMP's processor core passes values through the register file as it would occur in a real microprocessor, executing instructions out-of-order in data dependence fashion, and dispensing with SimpleScalar's functional in-order instruction execution.

SimpleMP implements a FIFO write-buffer containing committed store instructions until they have obtained coherence permissions and can be written to the cache, allowing the processor core to continue retiring instructions without blocking on store misses. SimpleMP also provides an aggressive sequential consistency implementation including exclusive prefetching and speculative load reordering as described by Gharachorloo et al. [40].

SimOS is a complete machine simulation environment consisting of simulators for the major components of a computer system (CPUs, memory hierarchy, disks, console, ethernet). We use a version of SimOS that simulates PowerPC-based computer systems running the AIX 4.3 operating system [52]. Communication between application code and I/O devices or the host machine is performed using a magic *sim_support* instruction that is embedded in modified versions of AIX device drivers. The PowerPC version of SimOS includes two processor simulators: the *simple* simulator, a straightforward one-instruction-at-a-time functional model, and the *block* simulator, a fast functional simulator making use of direct execution to achieve an order of magnitude improvement in simulation speed. We augment SimOS with a third processor model based on the SimpleMP simulator.

Figure 3-2 illustrates the steps typically involved when using this infrastructure, from checkpoint creation to checkpoint/PHARMsim validation to its usage as a timing simulator. Because of its simulation speed, we utilize the block-mode simulator to efficiently setup and tune workloads within the simulator. Once a workload is ready, a checkpoint of the system is created consisting of the system's memory, register, and I/O device state, as well as a log of the most recently

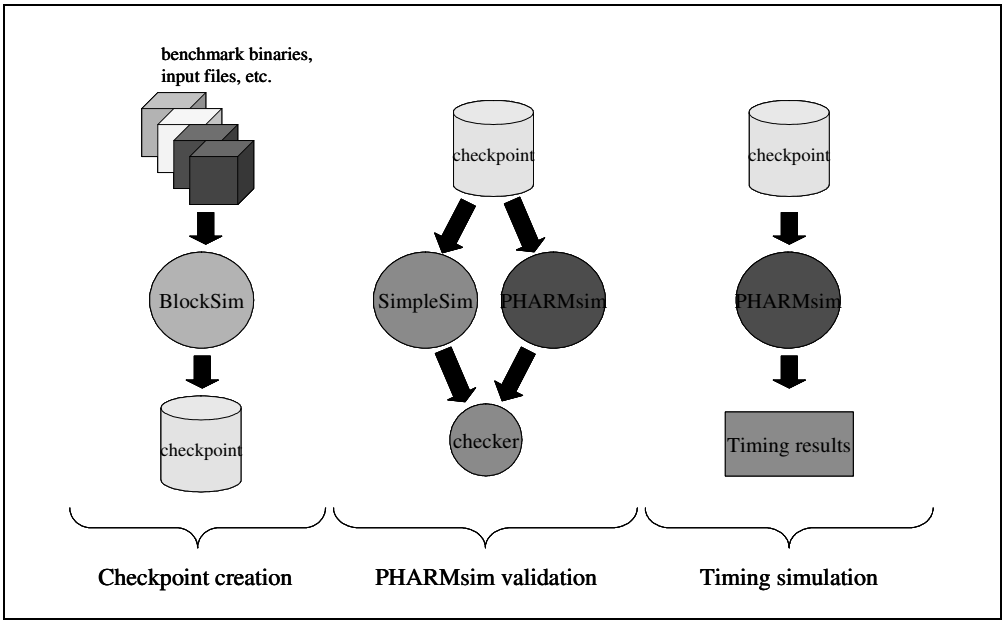

FIGURE 3-2. Typical PHARMsim usage flow. From checkpoint creation, to validation, to timing simulation.

referenced cache blocks. When starting a simulation from this checkpoint, the simulator's state is initialized using this snapshot, including warming the caches from the cache block log if the simulated machine includes a cache hierarchy. During the validation stage, we run the Simple simulator and PHARMsim in parallel, starting from the same checkpoint, and compare the differences in architected state that occur at instruction boundaries to ensure that PHARMsim can correctly execute this checkpoint. The Simple simulator is used because it executes instructions one-at-a-time (unlike the block mode simulator), allowing us to easily read architected state after each instruction has been executed. Once we have confidence that the checkpoint is correctly executed, we are able to perform performance studies using PHARMsim. The original version of SimpleMP could not have functioned properly in this environment without significant modifications, which we describe in the next section.

3.1.4 PHARMsim enhancements

In this section, we describe modifications to the SimpleMP simulator necessary for functionally supporting the PowerPC architecture and system-level code, and more accurate performance modeling.

Functional modeling:

- *PowerPC Instruction Set Architecture*: PHARMsim provides support for all of the instructions in the PowerPC instruction set, including privileged mode instructions. These instructions include the *dcbz* (data cache block zero), *dcbf* (data cache block flush), *dcbi* (data cache block invalidate), and *icbi* (instruction cache block invalidate) instructions which are fully supported by the two memory systems, as well as the *tlbie* (invalidate TLB entry) and *tlbia* (invalidate all TLB entries) instructions.
- *Address Translation*: PHARMsim fully implements the 32-bit address translation mode specified by the PowerPC architecture. On a TLB miss, a simulated hardware page-table walker issues a series of memory references to the simulated memory system, traversing the page table to find the correct mapping. If no mapping is found or a protection violation occurs, the simulated machine drains the pipeline, squashes prior in-flight instructions, and redirects fetch to the appropriate exception handler.
- *Precise Exceptions*: PHARMsim provides support for precise exceptions, whether they are caused by instructions or asynchronous interrupts. A micro-operation (see instruction cracking below) cannot commit until all micro-operations in the corresponding PowerPC instruction are known to

be exception-free (excluding certain load and store string instructions that PowerPC specifies are restartable). In the event of an instruction exception, all instructions in program order before the excepting instruction are allowed to commit, after which the pipeline is squashed and fetch is redirected to the appropriate exception handler. This support is overloaded for the implementation of the PowerPC instructions *rfi* (return from interrupt) and *isync* (instruction synchronize), which dictate that all instructions in the pipeline must be discarded before the instruction completes. In the event of an asynchronous exception, instruction fetch is stalled until the pipeline drains, and then redirected to the appropriate handler.

- *Reference and Change Bits*: In order to support virtual memory, PowerPC provides reference and change bits as part of the page table. When performing address translation for a memory operation, copies of the R/C bits stored in the TLB are inspected. If the corresponding bits are not already set for a particular reference, that instruction is flagged, and at commit the instruction must stall until a store marking the R/C bits has been performed to the appropriate page table entry.
- *Unaligned Memory References*: Unaligned memory references (which are allowed in the PowerPC architecture) are handled in the processor core by splitting each reference crossing a cache block boundary into two smaller aligned references which are then each issued to the SimpleMP memory system.
- *PowerPC Memory Consistency Model*: In addition to the implementations

of sequential consistency and TSO provided by SimpleMP, we have augmented the simulator with an implementation of PowerPC's weakly ordered model. To implement weak ordering, execution synchronizing instructions (e.g., *sync*) stall the dispatch pipeline stage until all prior load instructions have received their data and all prior store instructions have been ordered (including those in the write-buffer). The processor's load queue is also snooped, as in the base SimpleMP TSO implementation, however only same-address coherent load-to-load ordering ($\overrightarrow{rdrd-sa}$) is enforced. Other speculative load operations whose addresses conflict with incoming snoops are not squashed.

Performance Modeling:

SimpleMP lacks performance modeling for many common features of modern high-performance computer systems. In this section, we describe some of the enhancements made to more accurately model modern microarchitectures.

- *Instruction Cracking*: The PowerPC instruction set is supported through the addition of a "xlate" translation stage immediately prior to the dispatch stage. The xlate stage cracks PowerPC instructions into simpler micro-operations, similar to the workings of the IBM Power4 [107]. This instruction-cracking mechanism is further described in prior work [18].
- *Separate Scheduling Window*: Rather than using a unified RUU-based instruction scheduling window and reorder buffer [103] like SimpleScalar, PHARMsim models a separately sized scheduling window.
- *Deep Pipelines:* In order to model the increasing pipeline depths found in

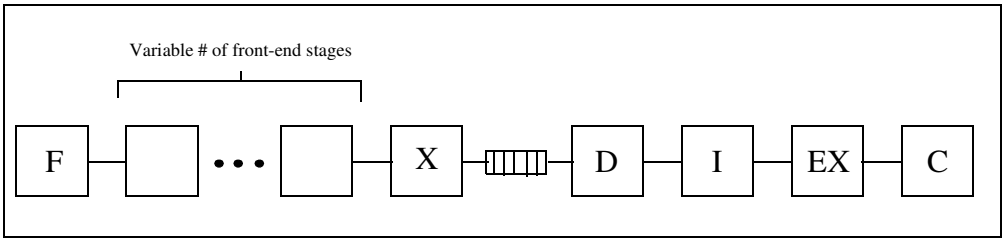

FIGURE 3-3. PHARMsim pipeline structure.

recent microprocessors, PHARMsim includes a user-controlled number of front-end pipeline stages between fetch and xlate. Once past the xlate stage, micro-operations commit in a minimum three cycles plus functional unit latency. Figure 3-3 illustrates the pipeline.

- *Memory Dependence Prediction:* By default, SimpleScalar and SimpleMP prevent ready load instructions from issuing until all prior store addresses have been calculated, which has been shown to be a serious performance detriment for many applications [81]. PHARMsim supports four different memory disambiguation modes*:* 1) *always-conflict*, which predicts that loads will always conflict and refrains from speculatively issuing them (as in the default SimpleScalar); 2) *never-conflict*, which always assumes that loads do not conflict and speculatively issues them, squashing incorrect speculations when a conflicting store address is calculated; 3) *store-set*, which uses a store-set predictor to predict those loads that conflict with prior outstanding stores [26]; and 4) *alpha*, which is identical to the Alpha 21264 memory disambiguation mechanism [28], utilizing a simple table containing load PC's that have caused ordering violations in the past, and preventing subsequent loads from speculatively issuing if there is a corresponding entry in the table.

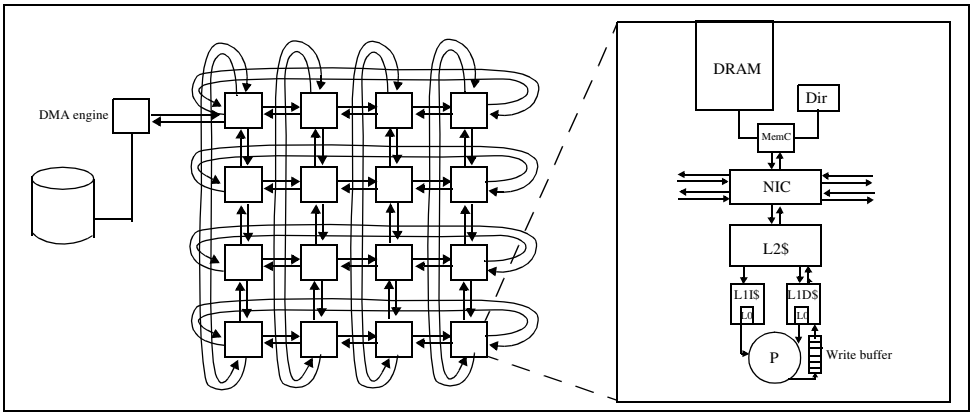

FIGURE 3-4. 16-processor directory-based NUMA memory system.

- *Level-Zero Cache*: PHARMsim augments the two-level cache hierarchy implemented by SimpleMP with a level-zero cache consisting of a subset of the level-one cache blocks. This addition allows the modeling of a three-level cache hierarchy without redesigning the existing coherence protocols.
- *Hardware Prefetcher*: PHARMsim implements a sequential hardware data prefetcher functionally equivalent to the Power4 hardware prefetcher [107]. Logic in the prefetcher detects streaming accesses that sequentially touch adjacent cache blocks (in either ascending or descending direction). Once a stream has been detected, an entry is allocated in an eight-entry stream buffer, which prefetches the next five cache blocks in the stream into the L2 cache, and the next adjacent block into the L1 cache. Prefetch streams terminate on 4k page boundaries.
- *Two-Dimensional Torus Network*: When using the NUMA system configuration provided by SimpleMP, PHARMsim replaces the fully-connected interconnection network with a statically dimension-order routed two-dimensional torus network consisting of three virtual channels. Figure 3-4 illustrates this configuration.

- *Direct Memory Access (DMA) Engine*: Both SimpleMP and SimpleScalar perform I/O "magically" by proxying system calls and instantaneously updating a cache's contents to reflect the new memory contents. We augment the SimpleMP memory systems with a coherent I/O controller device that functions as another agent in the coherence protocol. The SimOS I/O device models have been modified to use this DMA device when reading or writing memory.

3.2 Benchmarks

For performance evaluation, we use a variety of benchmarks, summarized in Table 3-1. During uniprocessor experiments, we use the SPEC 2000 benchmark suite and a few commercial workloads (TPC-B, TPC-H, and SPECjbb2000). For multiprocessor experiments, we use the SPLASH-2 parallel benchmark suite [111], SPECweb99, SPECjbb2000, TPC-B and TPC-H. For the parallel experiments presented in Chapter 5, our SPLASH-2 checkpoints are limited to barnes, ocean, radiosity, and raytrace because we do not have 16-processor checkpoints for the others. For the parallel experiments presented in Chapter 7, we present data using only our four-processor checkpoints only due to the excessive simulation time required to run the 16-processor checkpoints.

The SPEC integer and SPLASH-2 benchmarks were compiled with the IBM xlc optimizing C compiler, except for the C++ benchmark eon, which was compiled using g++ version 2.95.2. The SPEC floating point benchmarks were compiled using the IBM xlf optimizing Fortran compiler. The SPEC CPU bench-

Table 3-1: Benchmark descriptions.

Benchmark(s)	Comments
barnes	SPLASH-2 N-body simulation (8k particles)
cholesky	SPLASH-2 blocked sparse factorization kernel (input tk15.0)
fft	SPLASH-2 fast-fourier transform kernel (256k points)
fmm	SPLASH-2 N-body simulation in two dimensions (16k particles)
lu	SPLASH-2 dense matrix factorization kernel (512 x 512 matrix)
ocean	SPLASH-2 ocean simulation (514 x 514)
radiosity	SPLASH-2 light interaction application (-room -ae 5000.0 -en -0.0050 -bf 0.10)
radix	SPLASH-2 integer radix sort kernel (1M integers, radix 1024)
raytrace	SPLASH-2 raytracing application (car)
volrend	3D volume rendering using raycasting application. (head input)
water-nsquared	SPLASH-2 molecular physics application (512 molecules)
water-spatial	SPLASH-2 improved version of water-nsquared (512 molecules)
SPEC CPU 2000	Uniprocessor benchmarks from SPEC
SPECjbb2000	Server-side Java benchmark (IBM jdk 1.1.8 w/ JIT, 400 operations)
SPECweb99	Zeus Web Server 3.3.7, servicing 300 HTTP requests
TPC-B	Transaction Processing Council's original OLTP benchmark (IBM DB2 v 6.1)
TPC-H	Transaction Processing Council's decision support benchmark (IBM DB2 v. 6.1, running query 12 on a 512 MB database)

marks were run to completion using the MinneSPEC reduced input sets [53].

3.3 Insuring Comparable Results in Light of Non-determinism

As originally demonstrated by Alameldeen and Wood, the non-deterministic nature of multithreaded workloads can cause problems when comparing the performance of two machine configurations [6]. Should the amount of variance

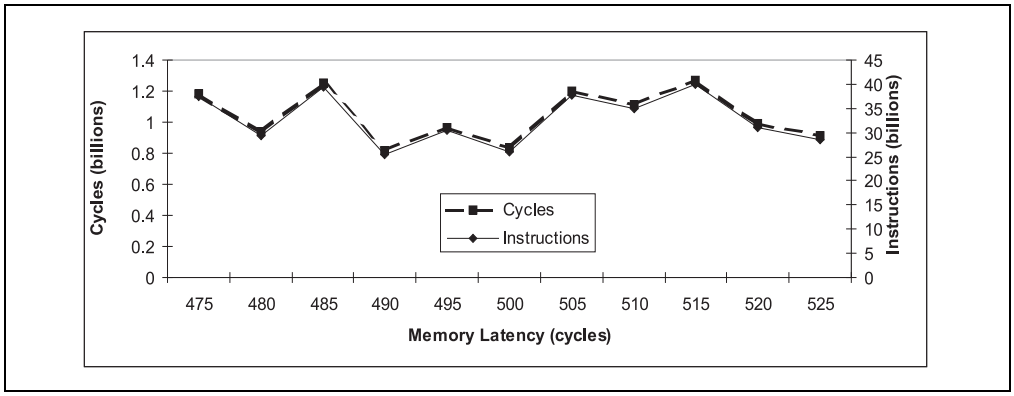

FIGURE 3-5. End-to-end simulated cycles (SPECjbb2000) for varying main memory latency. Main memory latency is varied from 475 to 525 cycles for a 16 processor run of SPECjbb2000 for 400 transactions starting from the same checkpoint. The measured number of cycles and instructions to complete the run is indicated, showing substantial variation in performance measurement for a minor architectural change

that exists within the workload due to timing dependent behavior be greater than the expected performance difference between the two machine configurations, it will be difficult to discern if the observed performance difference is caused by the differing machine configurations or the inherent workload variability. An example of this phenomenon is shown in Figure 3-5, graphing the execution time of the SPECjbb2000 benchmark for a certain machine configuration while varying main memory latency from 475 to 525 cycles. Although one would expect that as memory latency increases the benchmark execution time should also increase, in this case the execution time shows little correlation to memory latency. We also graph the dynamic instruction count from each run, and find that the dynamic instruction count also varies wildly, and that execution time roughly tracks dynamic instruction count.

In order to solve this problem, we use the statistical simulation approach advocated by Alameldeen and Wood [6]. When measuring the performance of multiprocessor systems, we perform multiple runs for each experimental data point while randomly inserting very small perturbations to the main memory latency

(between 0 and 4 cycles, less than 1% of main memory latency), causing the executions to diverge. When plotting results, we include error bars signifying a 95% confidence interval in the reported data. Although the non-determinism problem can also occur in single processor systems due to timing dependent interactions with I/O devices and timer interrupts, we have found such effects to be negligible for the workloads used in this thesis.

Lepak et al. describe an alternative methodology for ensuring comparable results based on forcing each execution to deterministically follow the same committed instruction stream [64]. Unfortunately, this method cannot be used to accurately evaluate techniques that depend on the ability to resolve races differently between the two executions, such as the edge-chasing delayed consistency mechanism described in Chapter 6. Consequently, we do not use the deterministic simulation methodology in this thesis. However, due to the inherent variability of the workloads excessive simulation time has been needed to collect the data in this thesis using the statistical simulation methodology. Some benchmarks have required 20 simulations per data point, while still garnering a margin of error of +/-3%. When a simulation may require several days of compute time, this methodology is not tractable for extensive design space exploration without the benefit of very large clusters for providing simulation bandwidth. We consequently advocate further research into the non-determinism problem, to either develop a high-precision performance methodology that does not require excessive simulation bandwidth, or provide a better understanding of the workload setup parameters that affect a workload's exhibited amount of non-determinism.

Chapter 4

Value-based Memory Ordering

Computer performance may be increased by improving any of the three terms of the fundamental performance equation: *CPU time = instructions/program x cycles/instruction x clock cycle time*. Unless a new instruction set is in development, improving the instructions/program term is largely beyond the architect's reach, and instead depends on the application developer or compiler writer. Hardware designers have therefore focused much energy on optimizing the latter two terms of the performance equation. Instructions-per-cycle (IPC) has increased by building out-of-order instruction windows that dynamically extract independent operations from the sequential instruction stream which are then executed in a dataflow fashion. Meanwhile, architects have also attempted to minimize clock cycle time through increased pipelining and careful logic and circuit design. Unfortunately, IPC and clock frequency are not independent terms. The hardware structures (e.g. issue queue, physical register file, load/store queues, etc.) used to find independent operations and correctly execute them out of program order are often constrained by clock cycle time. In order to decrease clock cycle time, the size of these conventional structures must usually decrease, also decreasing IPC. Conversely, IPC may be increased by increasing their size, but this also increases their access time and may degrade clock frequency. In this chapter, we focus on the scalability of one of these hardware structures: the load queue.

Load queues are usually built using content-addressable memories that

provide an address matching mechanism for enforcing the correct dependences among memory instructions. When a store address is generated, the load queue CAM is searched for prior loads that incorrectly speculatively issued before the store instruction. Depending on the supported consistency model, the load queue may also be searched when every load issues [28], upon the arrival of an external invalidation request [49][113], or both [107]. As the instruction window grows, so does the number of in-flight loads, resulting in the load queue CAM access latency increasing.

In this chapter, we present a mechanism called *value-based replay* that completely eliminates the associative search functionality requirement from the load queue in an attempt to simplify the execution core. Instead, memory dependences are enforced by simply re-executing load operations in program order prior to commit, and comparing the new value to the value obtained when the load first executed. We refer to the original load execution as the *premature load* and the re-execution as the *replay load*. If the two values are the same, then the premature load correctly resolved its memory dependences. If the two values differ, then we know that a violation occurred either due to an incorrect reordering with respect to a prior store or a potential violation of the memory consistency model. Instructions that have already consumed the premature load's incorrect value must be squashed. By eliminating the associative search from the load queue, we remove one of the factors that limits the size of a processor's instruction window. Instead, loads can reside in a simple FIFO either separately or as part of the processor's reorder buffer. In Section 4.2, we describe in detail value-based replay's impact on

the processor core implementation.

In order to mitigate the costs associated with replay (increased cache bandwidth and resource occupancy), we evaluate several heuristics that filter the set of loads that must be replayed. These filters, based on observations made using the constraint graph, reduce the percentage of loads that must be replayed exceptionally well (to 0.02 replay loads per committed instruction on average), and as a result, there is little degradation in performance when using load replay compared to a machine whose load queue includes a fully associative address CAM. Chapter 5 presents a thorough performance evaluation of value-based memory ordering using these filtering mechanisms.

Although the goal of value-based replay is eliminating associative lookup hardware from the load queue, the store queue also requires an associative lookup that will suffer similar scalability problems in large-instruction window machines. We focus on the load queue for three primary reasons: 1) because loads occur more frequently than store instructions (loads and stores constitute 30% and 14%, respectively, of dynamic instructions for the workloads in this thesis), the load queue in a balanced-resource machine should be larger than the store queue and therefore its scalability is more of a concern; 2) the store-to-load data forwarding facility implemented by the store queue is more critical to performance than the rare-error checking facility implemented by the load queue, and therefore the store queue's use of a fast search function is more appropriate; and 3) in some microarchitectures [28][107], the load queue is searched more frequently than the store queue, thus consuming more power and requiring additional read ports.

To summarize, this chapter makes the following contributions:

- **Elimination of associative search logic from the load queue via value-based replay**: We demonstrate a value-based replay mechanism for enforcing uniprocessor and multiprocessor ordering constraints that eliminates the need for associative lookups in the load queue.
- **Replay-reduction heuristics:** We introduce several novel heuristics that reduce the cache bandwidth required by the load-replay mechanism to a negligible amount.
- **Consistency model checking:** we define the constraints for implementing a back-end memory consistency checker. These constraints are also useful in the domain of other checking mechanisms such as DIVA [10]. Recent work has exposed a subtle interaction between memory consistency and value prediction that results in consistency violations under different forms of weak consistency [71]. Our value-based replay implementation may be used to detect such errors.

In the next section, we describe the microarchitecture of conventional associative load queues. Section 4.2 presents our alternative memory ordering scheme, value-based replay, including the description of our replay filtering heuristics. Using the experimental methodology described in Chapter 3, we present a detailed performance study of the value-based replay mechanism relative to an aggressive conventional load queue design in Chapter 5.

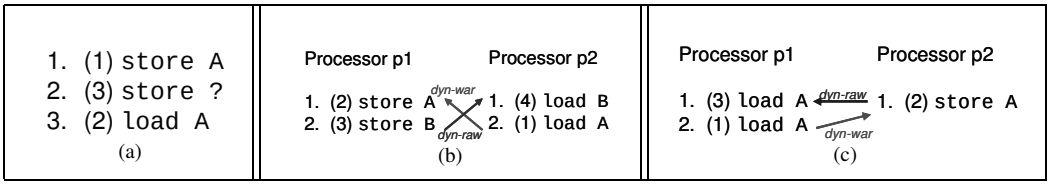

FIGURE 4-1. Correctly supporting out-of-order loads: Examples (a) uniprocessor RAW hazard, (b) multiprocessor violation of sequential consistency (c) multiprocessor violation of coherence

4.1 Associative Load Queue Design

We believe that an inordinate amount of complexity in an out-of-order microprocessor's design stems from the load queue, mainly due to an associative search function whose primary function is to detect rare errors. In this section, we examine the source of this complexity through an overview of the functional requirements of the load queue structures, and a description of their logical and physical design.

4.1.1 Functional requirements and logical design

The correctness requirements enforced by the load queue are two-fold: loads that are speculatively reordered with respect to a prior store that has an unresolved address must be checked for correctness, and violations of the multiprocessor memory consistency model caused by incorrect reorderings must be disallowed. Figure 4-1 (a) contains a code segment illustrating a potential violation of a uniprocessor RAW hazard. Each operation is labeled by its program order and by issue order (in parentheses). In this example, the load instruction speculatively issues before the previous store's address has been computed. Conventional memory ordering implementations enforce correctness by associatively searching the load queue each time a store address is computed. If the queue contains an already-

issued prior load whose address overlaps the store, the load is squashed and re-executed. In this example, if the second store overlaps address A, the scan of the load queue will result in a match and the load of A will be squashed.

Some microarchitectures delay issuing a load instruction until all prior store addresses are known, altogether avoiding the need to detect RAW dependence violations. Unfortunately this solution is not sufficient to prevent violations of the memory consistency model, which can occur if any memory operations are reordered. Forcing all memory operations to execute in program order is too restrictive, so associative lookup hardware is still required even if loads are delayed until all prior store addresses are known.

In terms of enforcing the memory consistency model, load queue implementations can be categorized into two basic types: those in which external invalidation requests search the load queue, and those in which the queue is not searched. We refer to these types as *snooping load queues* and *insulated load queues*. In a processor with a snooping load queue, originally described by Gharachorloo et al. [40], the memory system forwards external write requests (i.e. invalidate messages from other processors or I/O devices) to the load queue, which searches for already-issued loads whose addresses match the invalidation address, squashing any overlapping load. If inclusion is enforced between the load queue and any cache, replacements from that cache will also result in an external load queue search. Insulated load queues enforce the memory consistency model without processing external invalidations, by squashing and replaying loads that may have violated the consistency model. The exact load queue implementation will

depend on the memory consistency model supported by the processor, although either of these two types may be used to support any consistency model, as described shortly.

Figure 4-1(b) illustrates a multiprocessor code segment where processor *p2* has reordered two loads to different memory locations that are both written by processor *p1* in the interim. In a sequentially consistent system, this execution is illegal because all of the operations cannot be placed in a total order. A snooping load queue detects this error by comparing the invalidation address corresponding to *p1*'s store A to the load queue addresses, and squashing any instructions that have already issued to address A, in this case *p2*'s second load. An insulated load queue prevents this error by observing at load B's completion that the load A instruction has already completed, potentially violating consistency, and the load A is subsequently squashed. Loads at the head of the load queue are inherently correct with respect to the memory consistency model, and are therefore never squashed due to an external invalidation. Avoiding these squashes is important in order to ensure forward progress.

Processors that support strict consistency models such as sequential consistency and processor consistency do not usually use insulated load queues, due to the large number of operations that must be ordered with respect to one another (i.e. all loads). Insulated load queues are more prevalent in weaker consistency models, where there are few consistency constraints ordering instructions. For example, in the variant of weak ordering supported by the Alpha ISA, only those operations that are separated by a memory barrier instruction, or those operations

that read or write the same address (connected by $\overrightarrow{rdrd-sa}$ in the constraint graph), must be ordered. The Alpha 21264 supports weak ordering by stalling dispatch at every memory barrier instruction (enforcing the first requirement), and uses an insulated load buffer to order those instructions that read the same address [28]. Using the example in Figure 4-1(c), if processor p1's first load A reads the value written by p2, then p1's second load A must also observe that value. An insulated load buffer enforces this requirement by searching the load queue when each load issues and squashing any prior load to the same address that has already issued. Snooping load queues in sequentially consistent systems are simpler in this respect, because load instructions do not search the load queue, however this additional simplicity is offset by requiring support for external invalidation searches.

In order to reduce the frequency of load squashes, the IBM Power4 uses a hybrid approach that snoops the load queue, marking (instead of squashing) conflicting loads. Every load must still search the load queue at issue time for prior loads to the same address, however only those that have been marked by a snoop hit must be squashed [107].

Obviously, both the snooping and insulated load queue implementations are conservative in terms of enforcing correctness. Due to false sharing and silent stores [65], a load does not need to be squashed simply because its address matches the address of an external invalidation. The premature load may have actually read the correct value. Likewise, due to store value locality, when a store's address is computed, all prior loads to the same address that have already executed do not necessarily need to be squashed, since many may have actually read the cor-

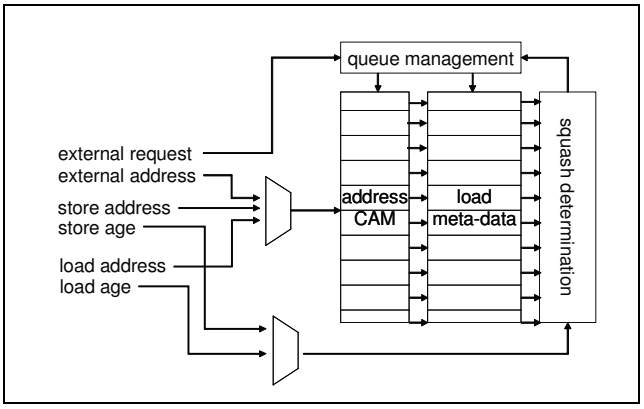

FIGURE 4-2. A simplified hybrid load queue.

rect value. These observations expose one benefit of the value-based ordering scheme, which has been exploited by other store-set predictor designs: a subset of squashes that occur in conventional designs are eliminated [85][115]. We quantify the frequency of avoided squashes in Chapter 5.

4.1.2 Physical design

Load queues are usually implemented using two main data structures: a RAM structure containing a set of entries organized as a circular queue, and an associated CAM used to search for queue entries with a matching address. The RAM contains meta-data pertaining to each dynamic load (e.g. PC, destination register id, etc.), and is indexed by instruction age (assigned in-order by the front-end of the pipeline). Figure 4-2 illustrates a simplified hybrid load queue with a lookup initiated by each load or store address generation (agen) and external invalidation. For address generation lookups, an associated age input is used by the squash logic to distinguish those loads that follow the load or store. The latency of searching the load queue CAM is a function of its size and the number of read/write ports. Write port size is determined by the processor's load issue width; each issued load must store its newly generated address into the appropriate CAM

Table 4-1: Load queue attributes for current dynamically scheduled processors.

Processor	Est. # read ports	Est. # write ports
Compaq Alpha 21364 (32-entry load queue, max 2 load or store agens per cycle)	2 (1 per load/store issued/cycle)	2 (1 per load issued/cycle)
HAL SPARC64 V (size unknown, max 2 loads and 2 store agens per cycle)	3 (2 for stores, 1 for external invalidations)	2
IBM Power 4 (32-entry load queue, max 2 load or store agens per cycle)	3 (2 for loads and stores, 1 for external invalidations)	2
Intel Pentium 4 (48-entry load queue, max 1 load and 1 store agen per cycle)	2 (1 for stores, 1 for external invalidations)	2

Table 4-2: Associative load queue search latency (nanoseconds), energy (nanojoules).

entries	Read/Write Ports (ns, nJ)			
	2/2	3/2	4/4	6/6
16	0.6 ns, 0.03 nJ	0.68 ns, 0.04 nJ	0.72 ns, 0.07 nJ	0.79 ns, 0.12 nJ
32	0.75 ns, 0.05 nJ	0.77 ns, 0.06 nJ	0.85 ns, 0.12 nJ	0.94 ns, 0.20 nJ
64	0.78 ns, 0.12 nJ	0.80 ns, 0.15 nJ	0.87 ns, 0.27 nJ	0.97 ns, 0.45 nJ
128	0.78 ns, 0.22 nJ	0.80 ns, 0.28 nJ	0.88 ns, 0.50 nJ	0.97 ns, 0.85 nJ
256	0.97 ns, 0.37 nJ	1.01 ns, 0.48 nJ	1.13 ns, 0.87 nJ	1.28 ns, 1.51 nJ
512	1.00 ns, 0.80 nJ	1.04 ns, 1.03 nJ	1.16 ns, 1.87 nJ	1.32 ns, 3.22 nJ

entry. The CAM must contain a read port for each issued store, each issued load (in weakly ordered implementations), and usually an extra port for external accesses in snooping load queues. A summary of their size in current generation processors having separate load queues (as opposed to integrated load/store queues) whose details have been published is found in Table 4-1, including an estimation of their read/write CAM port requirements. Typical current-generation dynamically scheduled processors use load queues with sizes in the range of 32-48 entries, and allow some combination of two loads or stores to be issued per cycle, resulting in a queue with two or three read ports and two write ports.

Using Cacti v. 3.2 [99], we estimate the access latency and energy per access for several CAM configurations in a 0.09 micron technology, varying the

number of entries and the number of read/write ports, as shown in Table 4-2. Although this data may not represent a lower bound on each configuration's access latency or energy (human engineers can be surprisingly crafty), we expect the trends to be accurate. The energy expended by each load queue search increases linearly with the number of entries, and the latency increases logarithmically. Increasing load queue bandwidth through multiporting also penalizes these terms: doubling the number of ports more than doubles the energy expended per access, and increases latency by approximately 15%. Based on these measurements, it is clear that neither the size nor bandwidth of conventional load queues scales well, in terms of latency or energy. These scaling problems will motivate significant design changes in future machines that attempt to exploit higher ILP through increased issue width or load queue size. In the next section, we present one alternative to associative load queues, which eliminates this CAM overhead.

4.2 Value-based Memory Ordering

The driving principle behind our design is to shift complexity from the timing critical components of the pipeline (scheduler/functional units/bypass paths) to the back-end of the pipeline. During a load's premature execution, the load is performed identically to how it would perform in a conventional machine: at issue, the store queue is searched for a matching address, if none is found and a dependence predictor indicates that there will not be a conflict, the load proceeds, otherwise it is stalled. After issue, there is no need for loads or stores to search the load queue for incorrectly reordered loads (likewise for external invalidations). Instead,

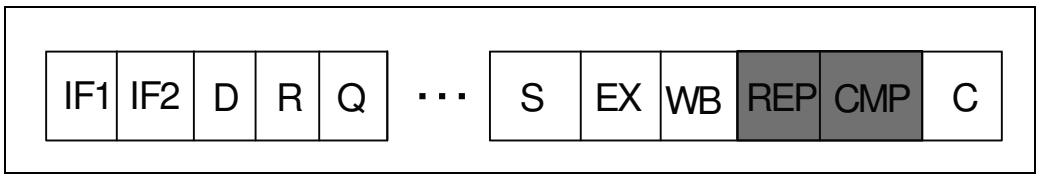

FIGURE 4-3. Pipeline diagram, replay stages highlighted.

we shift complexity to the rear of the pipeline, where loads are re-executed and their results checked against the premature load result. To support this load replay mechanism, two pipeline stages have been added at the back-end of the pipeline preceding the commit stage, labeled replay and compare and shown in Figure 4-3. For simplicity, all instructions flow through the replay and compare stages, with action only being taken for load instructions.

During the replay stage, certain load instructions access the level-one data cache a second time. We do not support forwarding from prior stores to replay loads, so load instructions that are replayed stall the pipeline until all prior stores have written their data to the cache. Because each replay load was also executed prematurely, this replay is less costly in terms of latency and power consumption than its corresponding premature load. For example, the replay access can reuse the effective address calculated during the premature load's execution, and in systems with a physically indexed cache the TLB need not be accessed a second time. In the absence of rare events such as an intervening cast-out or external invalidate to the referenced cache block between the premature load and replay load, the replay load will always be a cache hit, resulting in a low-latency replay operation. Because stores must perform their cache access at commit, the tail of the pipeline already contains datapath for store access. For the purposes of this work, we assume that this cache port may also be used for loads during the replay stage.

Loads compete with stores in the commit stage for access to the cache port, priority being given to stores.

During the compare stage, the replay load value is compared to the premature load value. If the values match, the premature load was correct and the instruction proceeds to the commit stage where it is subsequently retired. If the values differ, the premature load's speculative execution is deemed incorrect, and a recovery mechanism is invoked. The use of a conventional selective recovery mechanism is most likely precluded due to the variable latency and large distance between the processor's scheduling stage and replay stage, so we assume that a heavy-weight machine squash mechanism is used that squashes and re-executes all subsequent instructions, dependent and independent. At a high level, this replay mechanism can be viewed as an á la carte version of Austin's DIVA checker processor [10], checking only load instructions rather than all instructions.

Because the replay mechanism enforces correctness, the associative load queue can be replaced with a simple FIFO buffer that contains the premature load's address and data (used during the replay and compare stages), in addition to the usual meta-data stored in the load queue. To ensure a correct execution, however, care must be taken in designing the replay stage. The following three constraints guarantee any ordering violations are caught:

1) All prior stores must have committed their data to the L1 cache. This requirement ensures that RAW dependences are correctly satisfied for all loads. As a side-effect, it also eliminates the need for arbitration between the replay mechanism and store queue for access to the shared cache port; if there are

prior uncommitted loads in the pipeline, the store queue will not retire stores to the cache (due to typical handling of precise exceptions), and conversely, when there are prior uncommitted stores in the pipeline, the replay stage will not attempt to issue loads. Should this requirement result in performance degradation, it is possible that replay loads may also search the store queue, allowing them to replay while stores are still in the pipeline. However, we believe that this is unneeded complexity for current generation microarchitectures; in Chapter 5 we show that stalling replay loads until all prior stores have committed their data does not cause significant performance degradation.

2) All loads must be replayed in program order. To enforce consistency constraints, the local processor must not observe the writes of other processors out of their program order, which could happen if replayed loads are reordered. For the machine configuration used in the next chapter, we find that limiting replay to one instruction per cycle provides adequate replay bandwidth. Consequently, all loads are in-order because only one may be issued per clock cycle. In very aggressive machines, multiple load replays per cycle may prevent resource stalls. If all the replays performed in a single cycle are cache hits, their execution will appear atomic to other processors, and the writes of other processors will be observed in the correct order. In cases where a replay load causes a cache miss, correctness is ensured by forcing subsequent loads to replay after the cache miss is resolved.

3) A dynamic load instruction that causes a replay squash should not be replayed a second time after squash-recovery. This rule ensures forward progress in pathological cases where contention for a piece of shared data can per-

sistently cause a premature load and replay load to return different values.

Naively, all loads (except those satisfying rule 3) should be replayed to guarantee a correct execution. Unfortunately there are two primary costs associated with replaying loads, which we would like to avoid: 1) load replay can become a performance bottleneck given insufficient cache bandwidth for replays or due to the additional resource occupancy and 2) each replayed load causes an extra cache access and word-sized compare operation, consuming energy. In order to mitigate these penalties, we have investigated methods of eliminating the replay operation for certain dynamic load instructions. In the next two subsections, we define four filtering heuristics that are used to filter the set of loads that must be replayed while ensuring a correct execution. Filtered loads continue to flow through the replay and compare pipeline stages before reaching commit, however they do not incur cache accesses, value comparisons, or machine squashes. The first three filtering heuristics eliminate load replays while ensuring the execution's correctness with respect to memory consistency constraints. The final replay heuristic filters replays while preserving uniprocessor RAW dependences.

4.2.1 Filtering replays while enforcing memory consistency

The issues associated with avoiding replays while also enforcing memory consistency constraints are fairly subtle. To assist with our reasoning, we employ the constraint graph representation discussed in Chapter 2, using the example shown in Figure 4-4 in which processor *p1* incorrectly reads the original value of C. For the purposes of this discussion, we assume a sequentially consistent system, where there are four edge types: program order edges that order all memory opera-

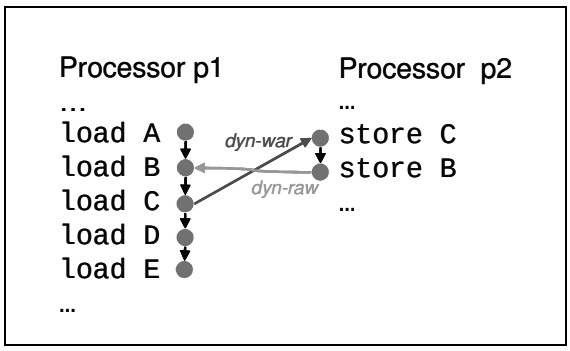

FIGURE 4-4. Constraint Graph Example.

tions executed by a single processor, and the dynamic ordering relations ($\overrightarrow{dyn-raw}$, $\overrightarrow{dyn-waw}$, $\overrightarrow{dyn-war}$) which order all memory operations that read or write the same memory location.

The following three replay filters detect those load operations that should be replayed to ensure correctness. The first two are based on the observation that any cycle in the constraint graph must include one of the dynamic ordering edges that connect instructions executed by two different processors. If an instruction is not reordered with respect to another instruction whose edge spans two processors, then there is no potential for consistency violation.

No-Recent-Miss Filter: One method of inferring the lack of a constraint graph cycle is to monitor the occurrence of cache misses in the cache hierarchy. If no cache blocks have entered a processor's local cache hierarchy from an external source (i.e. another processor's cache) while an instruction is in the instruction window, then there must not exist an incoming dynamic ordering edge from any other processor in the system to any instruction in the window. Consequently, we can infer that no cycle can exist, and therefore there is no need to replay loads to check consistency constraints. Using the example from Figure 4-4, the load B would incur a cache miss in an invalidation-based coherence protocol, requiring *p1*

to fetch the block from *p2*'s cache. When the block returns, *p1* would need to replay any load instructions currently in its instruction window. This filter may be implemented as follows: each time a new cache block enters a processor's local cache, the cache unit asserts a signal monitored by the replay stage. When this signal is asserted, a "recent miss/need-replay" flag is set to true and an age register is assigned the age index of the most-recently fetched load instruction in the instruction window. During each cycle that the flag is set to true, load instructions in the replay stage are forced to replay. After the flagged load instruction replays, if the age register still contains its age index, the flag is reset to zero.

No-Recent-Snoop Filter: The no-recent-snoop filter is conceptually similar to the no-recent-miss filter, only it detects the absence of an outgoing constraint graph edge, rather than an incoming edge. Outgoing edges can be detected by monitoring the occurrence of external write requests. If no blocks are written by other processors while a load instruction is in the out-of-order window, then there must not exist an outgoing $\overrightarrow{dyn-war}$ edge from any load instruction at this processor to any other processor. Reorderings across outgoing $\overrightarrow{dyn-raw}$ and $\overrightarrow{dyn-waw}$ edges are prevented by the in-order commitment of store data to the cache. When the no-recent-snoop filter is used, loads are only replayed if they were in the out-of-order instruction window at the time an external invalidation (to any address) was observed by the core. In terms of implementation, a mechanism similar to the no-recent-miss filter can be used. This heuristic will perform best in systems that utilize inclusive cache hierarchies, which filter the stream of invalidates observed by the processor. Because fewer invalidates reach the processor, fewer loads will

need to be replayed.

Care must be taken to ensure that the visibility of external invalidates is not lost due to castouts. For example, if a cache block is replaced and an invalidate arrives to that block immediately following the replacement, the invalidate will be filtered by the cache, and will not be sent to the processor. This exposes a window of opportunity during which a load that should be replayed is not. One solution to this problem (which we advocate) is to maintain an extra copy of the replaced cache block's tag until the oldest instruction in the out-of-order window retires. Incoming invalidates that match this tag are then forwarded to the processor, triggering a replay for load instructions in the instruction window.

No-Reorder Filter: The no-reorder filter is based on the observation that the processor often executes memory operations in program order. If so, the instructions must be correct with respect to the consistency model, therefore there is no need to replay any load. We can detect operations that were originally executed in-order using the instruction scheduler, by marking loads that issue while there are prior incomplete loads or stores.

4.2.2 Filtering replays while enforcing uniprocessor RAW dependences

In order to minimize the number of replays needed to enforce uniprocessor RAW dependences, we use the observation that most load instructions do not issue out of order with respect to prior unresolved store addresses. The *no-unresolved-store filter* identifies loads that did not bypass any stores with unresolved addresses when issued prematurely. These loads are identified and marked at issue time, when the store queue is searched for conflicting writes from which to for-

ward data. A similar filter was used by Park et al. to reduce the number of load instructions that must be inserted into the load queue [87].

4.2.3 The interaction of filters

Of the four filters described above, only the no-reorder filter can be used in isolation; each of the other three are too aggressive. The no-recent-snoop and no-recent-miss filters eliminate all replays other than those that can be used to infer the correctness of memory consistency, at the risk of breaking uniprocessor dependences. Likewise, the no-unresolved-store filter eliminates all replays except those used to preserve uniprocessor RAW dependences, at the risk of violating the memory consistency model.

Consequently, the no-unresolved-store filter should be paired with either the no-recent-snoop or no-recent-miss filters to ensure correctness. If the no-unresolved-store filter indicates that a load should be replayed, it is replayed irrespective of the consistency filter output. Likewise, if the consistency filter indicates that a load should be replayed, it is replayed irrespective of the no-unresolved-store filter. For further improvement, the no-recent-snoop filter and no-recent-miss filter can be used simultaneously. However we find that these filters work well enough in isolation that we do not explore this option. In the next chapter, we evaluate the value-based replay mechanism using these filters.

4.3 Discussion

In this section, we discuss the interaction of value-based replay with two common microarchitectural techniques: memory dependence prediction and

simultaneous multithreading. We conclude with a discussion of the potential for power-savings offered by the value-based memory ordering mechanism.

4.3.1 Interaction with memory dependence predictors

One drawback to the value-based replay mechanism is its inability to correlate a misspeculated load dependency to the store on which it depends. In a conventional design, store addresses search the load queue as they are computed to find dependent loads that erroneously issued prematurely. When a match is found, the identity of the store on which the load depends is unambiguous. However, when a load incurs a memory ordering violation in the value-based scheme, it is unclear which store caused the misspeculation, and whether that store was even performed by the local processor or a remote processor.

Some memory dependence mechanisms rely on the identification of the conflicting store for training, and thus cannot be properly trained if this identification cannot be performed. Dependence predictors of this type include Moshovos et al.'s original dependence predictor [81], store set predictors [26], store barrier caches [46] and the colliding store-distance predictor proposed by Yoaz et al. [114].

Consequently, when evaluating value-based memory-ordering in Chapter 5, we use a simple predictor that is functionally equivalent to the dependence predictor used in the Alpha 21264, where a PC-indexed table is used whose entries contain a single bit indicating whether the load at that PC has been a victim of a previous dependence misprediction [28]. If this bit is set, corresponding loads are prevented from issuing until the addresses of all prior stores are computed. For

fairness, our evaluation of value-based replay in Chapter 5 includes a comparison to a baseline machine incorporating a store-set predictor, showing a negligible difference in performance. Should this prediction strategy cause a performance degradation in more-aggressive microarchitectures, which may exploit a greater degree of instruction-level parallelism than the machine configuration used in Chapter 5, it may be possible to find alternative methods of training those dependence predictors which depend on conflicting store identification. For example, Yoaz et al.'s method uses a tagged prediction table indexed by load PC whose entries contain the distance between the load and colliding store. Loads that match an entry in the prediction table may issue once all stores at this distance or beyond have resolved their addresses and are found to be independent. If this predictor were included in a machine that uses value-based memory ordering, it could be trained by simply initially setting the distance to one, instead of the precise distance. Each time a load instruction causes a memory ordering violation, this distance would be incremented until the counter saturates or the violations cease. With the addition of path-based information used to index the table, this prediction strategy may rival that of a true store-set predictor.

4.3.2 Value-based memory ordering in simultaneously multithreaded processors

Our description of value-based memory ordering above has assumed a conventional single-threaded processor. The specifics of the memory ordering implementation found in a simultaneous multithreaded (SMT) processor may affect the functionality of value-based memory ordering and its associated filters. Unfortu-

nately, there have been no details published on how current-generation SMT implementations constrain load issuance in the presence of prior conflicting stores that belong to another thread. In this section we include a brief discussion our assumptions regarding SMT memory disambiguation, and describe how the value-based memory ordering mechanism functions in this organization.

For the purposes of this discussion, we assume a store queue that is shared among threads, where each load searches the store addresses corresponding to all threads prior to issuing. When a load issues, three store-queue address match scenarios exist:

- *definite inter-thread match*: the load conflicts with a prior store from another thread. In this case we assume the load will stall until the store data has been written to the L1 cache, so that if the store is squashed before reaching commit, the load cannot have read incorrect speculative data.
- *potential inter-thread match:* there are prior stores with unresolved addresses from another thread that may conflict. In this case, we assume loads speculatively issue (if stalling on unresolved addresses hurts performance in single-threaded processors, it will also hurt in multithreaded processors).
- *no inter-thread match:* there are no prior unresolved address stores or conflicting stores from another thread.

Based on these assumptions, the value-based replay mechanism would work as follows: in the no inter-thread match case, it would function identically to the non-SMT implementation. There is no need to replay a load unless one of the

other replay triggers described in Section 4.2 occurs (i.e. there is a potential interaction with another processor), because the load is guaranteed to receive the correct value with respect to stores issued from the local processor, regardless of which thread may issue them.

In the potential inter-thread match case, the replay mechanism and filters would also function identically to the non-SMT implementation: the no-unresolved store address filter would cause the load to be replayed because there is a prior store with an unresolved address at issue time, forcing the replay load to obtain the correct value from the cache in case the premature load read the incorrect value. In the definite inter-thread match case, instead of forwarding data from the store to the load (as would happen in a non-SMT implementation), the load is stalled until the store has written its data to the cache. When the premature load subsequently issues, it will be guaranteed to receive the correct data from the cache.

Other multi-threading implementations such as fine-grained multithreading (as implemented in the Tera computer system [8]) and coarse-grained multithreading (as implemented in the IBM RS-64 III processor [14]) have relied on in-order instruction issue within a single-thread, eliminating the need for explicit memory ordering detection/correction hardware.

4.3.3 Potential for power savings

Due to the growing importance in reducing microprocessor power consumption [83], there has also been much recent research exploring energy-saving alternatives to conventional microarchitectural structures. In terms of power con-

sumption, the value-based replay scheme should consume less power than conventional load queues, because it only requires at most O(number of dynamic loads) comparisons (one for cache tag match and one for value comparison) per dynamic load instruction. Conventional load queues require at least O(number of load queue entries) address comparisons per dynamic store instruction (plus dynamic load instructions if weak ordering, plus incoming invalidates if snooping load queue).

Extrapolating any quantitative power estimates would be unreliable without using a detailed underlying power model. Instead, one can get a rough estimate of the difference in dynamic energy using a simple model:

$$\begin{aligned}\Delta Energy = {} & ((E_{cacheaccess} + E_{wordcomparison}) \times replays) \\ & - (E_{ldqsearch} \times ldqsearches) + overhead_{replay}\end{aligned}$$

The primary energy cost for the value-based replay mechanism is the energy consumed by replay cache accesses and word-sized comparison operations. This cost is multiplied by the number of replays, which we have empirically determined to be relatively small. In the equation above, $overhead_{replay}$ includes the energy cost of the two extra pipeline latches and replay filtering mechanism. Because the number of replays is quite small (on average 0.02 replay loads per committed instruction using the no-recent-snoop/no-unresolved-store filter configuration, as shown in Chapter 5), if an implementation's load queue CAM energy expenditure per committed instruction is greater than 0.02 times the energy expenditure of a cache access and word-sized comparison, we expect the value-based replay scheme to yield a reduction in power consumption. Technology scaling is

increasing both the absolute and relative contribution of static power dissipation to overall power dissipation [108]. In terms of static power dissipation, value-based memory ordering should also be advantageous, because it should require less area than conventional memory ordering due to the elimination of the large address matching CAMs.

4.4 Related Work

There has been much recent research on scaling structures in ways that are amenable to high clock frequencies without negatively affecting IPC. Much of this work has focused on the instruction issue queue, physical register file, and bypass paths, but very little has focused on the load queue or store queue [4][87][97]. To prevent this access time from affecting a processor's overall clock cycle time, recent research has explored variations of conventional load/store queues that reduce the size of the CAM structure through a combination of filtering, caching, or segmentation. Sethumadhavan et al. employ bloom filtering of LSQ searches to reduce the frequency of accesses that must search the entire queue [97]. Bloom filters are also combined with an address matching predictor to filter the number of instructions that must reside in the LSQ, resulting in a smaller effective LSQ size. Akkary et al. explore a hierarchical store queue organization where a level-one store queue contains the most recent *n* stores, while prior stores are contained in a larger, slower level-two buffer [4]. A fast filtering mechanism reduces level-two lookups, resulting in a common-case load queue lookup that is the larger of the level-one store queue lookup latency and filter lookup latency. Their research does

not address the scalability of the load queue. Park et al. explore the use of a store-set predictor to reduce store queue search bandwidth by filtering those loads that are predicted to be independent of prior stores. Load queue CAM size is reduced by removing loads that were not reordered with respect to other loads, and a variable-latency segmented LSQ is also explored [87]. Although each of these proposals offers a promising solution to the load or store queue scalability problem, their augmentative approach results in a faster search but also adds significant complexities to an already complex part of the machine.

There has also been related work on low power alternatives to conventional load queues, by Ponomorev et al. who explore the power-saving potential of a segmented load queue design where certain portions of the load/store queue are disabled when occupancy is low. However, this work does not address the scalability problem of a fully occupied load queue [89].

Instead of the hardware mechanism presented here, Altman et al. use a mechanism similar to value-based memory ordering to ensure the correctness of instruction reorderings using software in the context of the DAISY binary translation system [7]. During aggressive optimization, loads may be hoisted above prior conflicting stores and reordered in ways that could violate the consistency model. To protect against potential violations while still allowing the reordering, a special *load-verify* instruction is inserted at the original site of the load instruction, which detects any incorrect reordering. Should such a violation occur, control is transferred to a software recovery routine.

Chapter 5

Experimental Evaluation of Value-based Memory Ordering

In this chapter, we present a detailed performance evaluation of the value-based replay memory ordering mechanism. We compare our mechanism to an aggressive eight-wide baseline machine (described in Section 5.1) to demonstrate that value-based replay is able to keep up with a high-performance conventional design, despite the replay-based machine's associated performance penalties. We find that the value-based replay mechanism is very competitive with the baseline machine, averaging less than one percent performance degradation across this set of benchmarks when using the best filter configuration, as shown in Section 5.2.

The baseline machine configuration utilizes a large unified load/store queue, under the assumption that it can be accessed in a single cycle, which will not be true as clock cycle times continue to decrease. In Section 5.3, we compare the value based replay mechanism to machine configurations with smaller, more realistically sized load queues. We find that when the CAM-based load queue size is constrained, value-based replay can provide performance benefit through the removal of this constraint.

Increased resource occupancy is primarily responsible for any performance degradation caused by value-based replay: all instructions must pass through two additional pipeline stages before freeing their associated resources, and some load instructions must access the cache, stalling the machine until all prior stores have committed and then for the duration of the cache access latency. The baseline

Table 5-1: Baseline machine configuration.

Out-of-order execution	5.0 GHZ, 15-stage 8-wide pipeline, 256 entry reorder buffer, 128 entry load/store queue, 32 entry issue queue, store-set predictor with 4k entry SSIT and 128 entry LFST (baseline only), 4k entry simple Alpha-style dependence predictor [28] (replay-based only).
Functional Units (latency)	8 integer ALUs (1), 3 integer MULT/DIV (3/12), 4 floating point ALUs (4), 4 floating point MULT/DIV (4, 4), 4 L1D load ports in OoO window, 1 commit stage L1D load/store port
Front-end	fetch stops at first taken branch in cycle, combined bimodal (16k entry)/gshare (16k entry) with selector (16k entry) branch prediction, 64 entry RAS, 8k entry 4-way BTB
Memory System (latency)	32k direct-mapped IL1 (1), 32k direct-mapped DL1 (1), 256k 8-way DL2 (7), 256k 8-way IL2 (7), Unified 8MB 8-way L3 (15), 64 byte cache lines. 2k entry 2-way itlb, 2k entry 2-way dtlb. Memory (400 cycles/100 ns best-case latency, 10 GB/S bandwidth), Stride-based prefetcher modeled after Power4

machine used in the first part of this evaluation includes a large, 256 entry reorder buffer, and an aggressive one cycle L1 data cache access latency, possibly masking any performance degrading effects due to resource occupancy. In Section 5.4, we evaluate the replay mechanism when assuming longer L1 data cache latencies, and in Section 5.5, when assuming a smaller reorder buffer. In both cases, we find that value-based replay is very competitive with the conventional machine configuration, within 1% of the conventional machine's performance on average.

5.1 Machine Configuration

Table 5-1 describes the machine configuration used for these experiments. The experimental machine model uses a load/store queue sized equivalently to the baseline machine. The baseline and experimental configurations use different dependence predictors due to value-based replay's inability to correlate dependence mispredictions with the conflicting store. The baseline machine uses a store-

set predictor [26], while the experimental configuration uses a PC-indexed table storing a single bit for loads that have caused a dependence misspeculation in the past, as in the Alpha 21264 [28].

For the multiprocessor performance data, we augment the machine configuration from Table 5-1 with a Sun Gigaplane-XB-like interconnection network for communication among processors and memory [24] that incurs an extra 32 cycle latency penalty for address messages and 20 cycle latency penalty for data messages. We assume a point-to-point data network in which bandwidth scales with the number of processors. For this multiprocessor machine configuration, we use a sequential consistency memory model. Although a weaker consistency model would reduce the number of required replays even further because there are fewer instructions that must be ordered with respect to one another, we find that the number of replays due to consistency constraints is already so low (shown in Figure 5-2) that further exploration is unwarranted.

5.2 Value-based Replay Relative to Large Conventional LSQ

Figure 5-1 presents the performance of value-based replay using four different filter configurations: no filters enabled (labeled replay all), the no-reorder filter in isolation, the no-recent-miss and no-unresolved-store filters in tandem, and the no-recent-snoop and no-unresolved-store filters in tandem. This data is normalized to the baseline IPC reported at the bottom of each set of bars.

The value-based replay mechanism is very competitive to the baseline machine despite the use of a simpler dependence predictor. Without any filtering

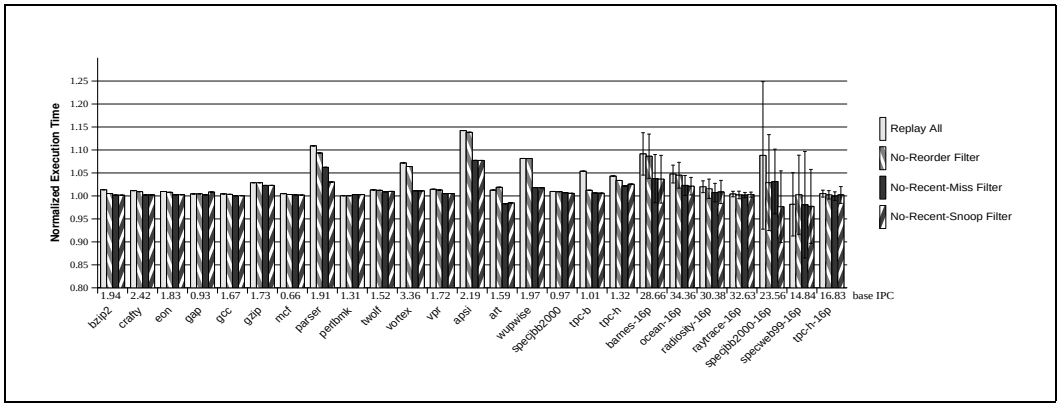

FIGURE 5-1. Value-based replay performance. (Relative to baseline)

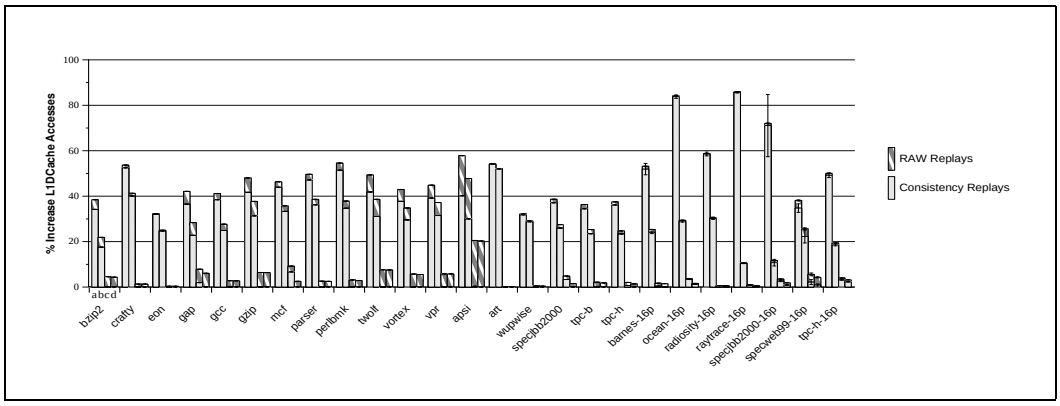

FIGURE 5-2. Increased data cache bandwidth due to replay. (a) replay all, (b) no-reorder filter, (c) no-recent-miss filter, (d) no-recent-snoop filter

mechanism, value-based replay incurs a performance penalty of only 3% on average. The primary cause of this performance degradation is an increase in reorder buffer occupancy. Figure 5-2 shows the increase in L1 data cache references for each of the value-based configurations. Each bar is broken into two segments: replays that occur because the load issued before a prior store's address was resolved, and replays that occur irrespective of uniprocessor constraints. Without filtering any replays, accesses to the L1 data cache increase by 49% on average, ranging from 32% to 87% depending on the percentage of cache accesses that are caused by wrong-path speculative instructions and the fraction of accesses that are stores. This machine configuration is limited to a single replay per cycle due to the single back-end load/store port, which leads to an increase in average reorder

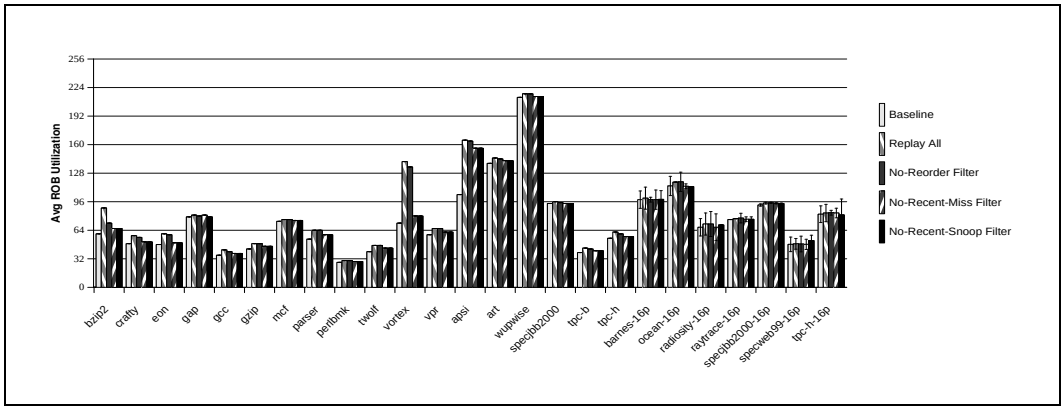

FIGURE 5-3. Average reorder buffer utilization.

buffer utilization due to cache port contention (most dramatically in apsi and vortex, as shown in Figure 5-3). This contention results in performance degradation due to an increase in reorder buffer occupancy and subsequent reorder buffer allocation stalls. Although this performance degradation is small on average, there are a few applications where performance loss is significant.

When the no-reorder filter is enabled, the performance of value-based replay improves, although not dramatically. The no-reorder filter is not a very good filter of replays, reducing the average cache bandwidth replay overhead from 49% to 30.6%, indicating that most loads do execute out-of-order with respect to at least one other load or store. The no-recent-snoop and no-recent-miss filters, when used in conjunction with the no-unresolved-store filter, solve this problem. For the single-processor machine configurations, there are no snoop requests observed by the processor other than coherent I/O operations issued by the DMA controller, which are relatively rare for these applications. Consequently, the no-recent-snoop filter does a better job of filtering replays than the no-recent-miss filter. This is also true in the 16-processor machine configuration, where an inclusive cache hierarchy shields the processor from most snoop requests. As shown in Figure 5-2, the

extra bandwidth consumed by both configurations is small, 4.3% and 3.4% on average for the no-recent-miss and no-recent-snoop filters respectively. The large reduction in replays leads to a reduction in average reorder buffer utilization (shown in Figure 5-3), which leads to an improvement in performance for those applications that were negatively affected in the replay-all configuration. For the single processor results, value-based replay with the no-recent-snoop filter is only 1% slower than the baseline configuration on average. For the multiprocessor configuration, the difference is within the margin of error caused by workload nondeterminism.

Many of the integer benchmarks whose resource occupancy was already low do not suffer much performance degradation due to increased resource occupancy, but are more affected by the change from the store-set predictor to the Alpha-style dependence predictor. Neither predictor incurs many memory order violations, but for some benchmarks the simple predictor more frequently stalls loads due to incorrectly identified dependences, ultimately decreasing IPC. In one case (art) the reverse is true, where the baseline machine's store-set predictor stalls a significant fraction of loads unnecessarily, resulting in a performance improvement in the value-based replay configurations. We attempted to exacerbate the negative effects of the simple dependence predictor relative to the store-set predictor by repeating these experiments using a larger 256-entry issue queue, but found that the results do not differ materially for this machine configuration.

One advantage of the value-based replay mechanism is its ability to filter dependence misspeculation squashes if the misspeculated load happens to return

the value of the conflicting store. Due to this value locality, we find that the value based mechanism on average eliminates 59% of the dependence misspeculation squashes caused by RAW uniprocessor violations, because the replay load happens to receive the same value as the premature load, even though the premature load should have had its data forwarded by an intermediate store. However, the frequency of squashes for these applications is so low (on the order of 1 per 100 million instructions), this reduction has little effect on overall performance.

The results are similar for consistency violations. In the multiprocessor machine configuration, value-based replay is extremely successful at avoiding consistency squashes, eliminating 95% on average. However we once again find that such machine squashes occur so infrequently (4 per 10,000 instructions in the most frequent case, SPECjbb2000) their impact on performance is insignificant. Should consistency squashes be a problem with larger machine configurations or applications where there is a greater level of contention for shared data structures, value-based replay is a good means of improvement.

5.3. Constrained Load Queue Size

The previous set of performance data uses a baseline machine configuration with a large, unified load/store queue. The primary motivation for value-based replay is to eliminate the large associative load queue structure from the processor, which does not scale as clock frequencies increase. Figure 5-4 presents a performance comparison of the best value-based replay configuration (the no-recent-snoop and no-unresolved-store filters) to a baseline machine configuration that uses a separate smaller load queue, for two different sizes, 16-entries and 32-entries. A 32-entry load queue is representative of current generation load queue

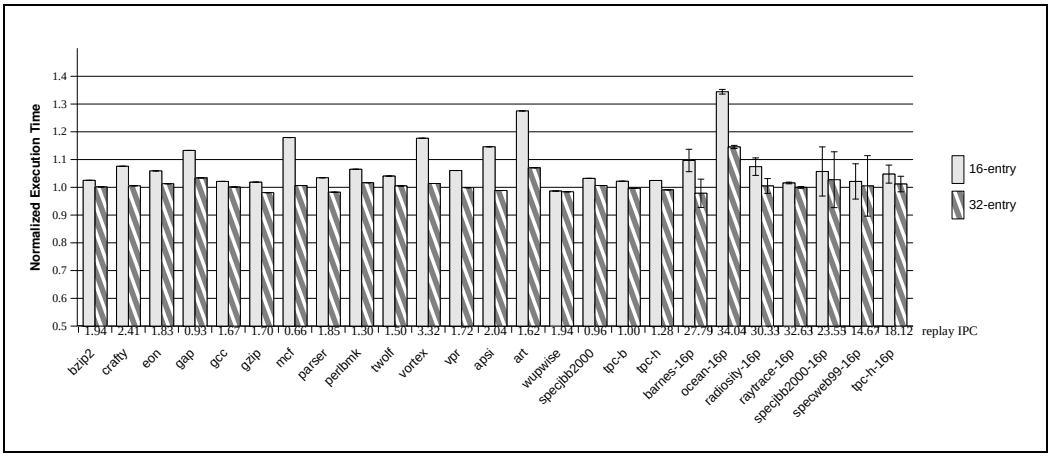

FIGURE 5-4. Performance of constrained load queue sizes. Relative to value-based replay with no-recent-snoop/no-unresolved-store filters.

sizes, and makes a fairly even performance comparison to the value-based replay configuration. On average, the value-based replay configuration is 1.0% faster, with art and ocean being significantly faster due to their sensitivity to load queue size (7% and 15% respectively). In future technology generations, a 32-entry load queue CAM lookup may not fit into a single clock cycle. When the load queue size is constrained to 16 entries, value-based replay offers a significant advantage, at most 34% and averaging 8% performance improvement.

5.4 Sensitivity to L1 Data Cache Latency

In this section, we evaluate value-based replay using two machine configurations with increased L1 data cache latency: two cycles and four cycles, shown in Figure 5-5 and Figure 5-6 respectively. In both cases the performance of value-based replay relative to the baseline machine is nearly identical to its relative performance using a single cycle cache. Because there are so few load instructions that must be replayed, it is rare for a load instruction to stall the pipeline. In many applications (apsi, art, gap, mcf, barnes, ocean, radiosity, raytrace, and tpc-h), the

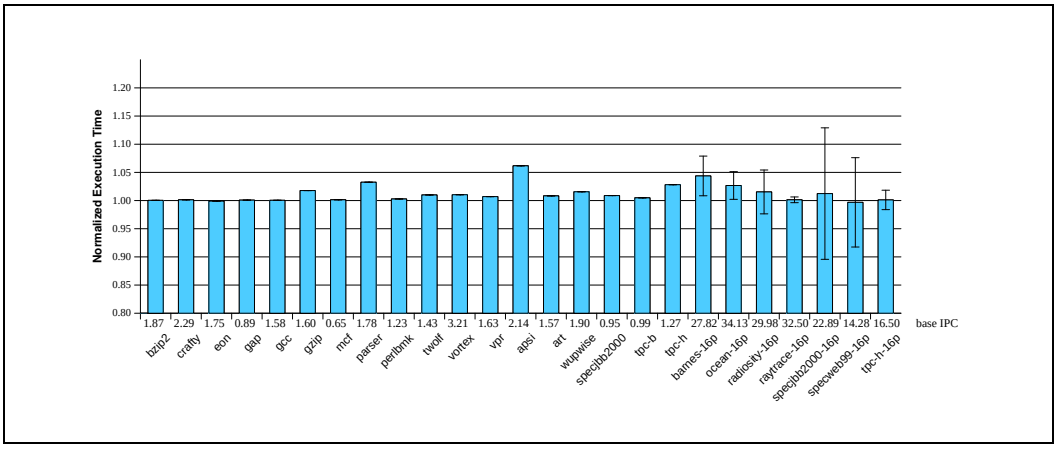

FIGURE 5-5. Value-based replay performance relative to baseline. (2 cycle L1 data cache latency)

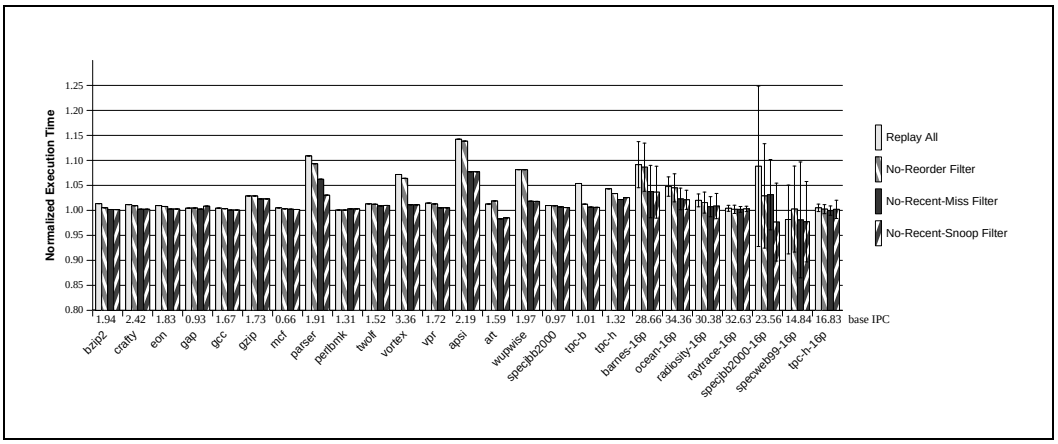

FIGURE 5-6. Value-based replay performance relative to baseline. (4 cycle L1 data cache latency)

average reorder buffer occupancy actually decreases when increasing the L1 data cache latency. Since the reorder buffer size is kept constant, this creates an extra cushion of reorder buffer that helps hide any extra replay latency.

When using a four-cycle cache latency, the relative performance of gap and parser improve. For gap, this is also caused by the simple dependence predictor stalling fewer loads than the store-set predictor. For parser, there is no single root cause of the performance difference. The branch predictor, simple dependence predictor, and data cache all performed slightly better than in the baseline execution, resulting in a small (3%) performance improvement.

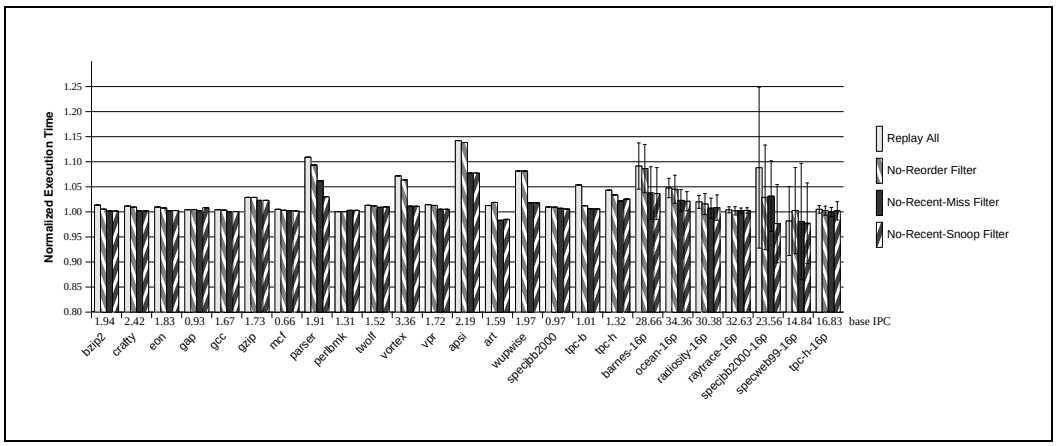

FIGURE 5-7. Value-based replay performance relative to baseline. (64 entry reorder buffer, 32 entry LSQ)

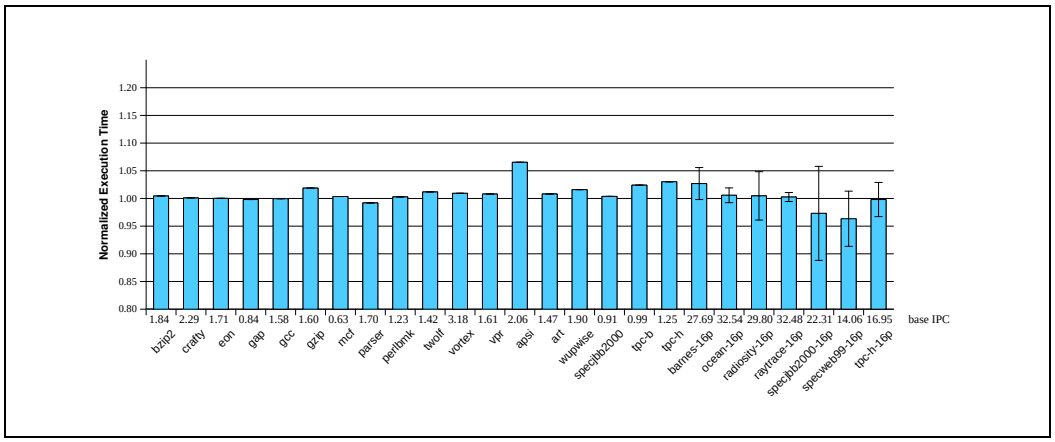

FIGURE 5-8. Value-based replay performance relative to baseline. (128 entry reorder buffer, 64 entry LSQ)

5.5 Sensitivity to Instruction Buffering Resources

Because value-based replay can increase resource occupancy, it is only fair that we also present performance results using a machine configuration with fewer resources than the 256 entry ROB, 128 entry LSQ machine used above. Figure 5-7 and Figure 5-8 show the performance of value-based replay relative to the baseline, using a machine with 64 ROB/32 LSQ and 128 ROB/64 LSQ respectively. For this data, we also use a more realistic two-cycle level one data cache latency.

As can be seen in the graphs, the data looks almost exactly like the previ-

ous sets of data, even when using a 64 entry ROB/32 entry LSQ, averaging 1% performance degradation across all applications. As the size of the instruction window shrinks, there are fewer instructions in flight so there are consequently fewer operations reordered, resulting in fewer replays being triggered. This effect reduces the number of replays by a minor amount moving from the 256 entry ROB to the 128-entry reorder buffer, but the reduction is significant with the 64-entry ROB, resulting in 13% fewer replays on average compared to the 256 entry ROB machine (34% fewer in gcc, the largest reduction). In the 64 entry ROB machine, the average reorder buffer occupancy increases by no more than two in any application other than apsi. Also, most applications have an average reorder buffer occupancy of less than 50, indicating that there is usually a cushion of a few entries to tolerate the latency penalty caused by value-based reordering.

5.6 Summary

In this chapter, we evaluated the performance of value-based replay relative to an aggressive microarchitecture using a large conventional load queue. We have shown that value-based replay is competitive in terms of performance, averaging 1% performance degradation across benchmarks using machine configurations with a variety of instruction window resources and data cache latencies. We consider this small of a reduction in performance to be more than offset by the reduction in complexity garnered by the elimination of the conventional CAM-based load queue. With value-based ordering, we have eliminated one obstacle to the creation of very large instruction windows by allowing load instructions to

reside in a simpler reorder buffer-like structure, rather than the CAM-based structure which does not scale well.

Additionally, we have shown that value-based replay requires very little extra cache bandwidth, resulting in an average increase of 3.5% across the applications studied here. Given this small number of replays (0.02 per committed instruction on average), we believe that value-based replay may pose a more energy-efficient alternative to conventional load queues in future microarchitectures as well.

The success of value-based replay has been enabled by the creation of several replay reduction heuristics that are quite successful at eliminating replays. The invention of these heuristics was spurred by our thinking about the problem in terms of the constraint graph, specifically ways in which an acyclic constraint graph can be inferred based on locally observable information. Although the topic of Chapter 4 and the performance analysis presented in this chapter have been focused on reducing the complexity of one small part of a processor's microarchitecture, this example illustrates the depths to which many aspects of a design are affected by the choice of memory consistency model and its implementation, and the benefits provided by a better understanding of the necessary requirements for supporting the model.

Chapter 6

Edge-Chasing Delayed Consistency: A New Implementation of Weak Ordering

There is a close relationship between the performance of shared-memory multiprocessors and the fraction of memory operations that can be satisfied from local cache hierarchies. Due to the growing disparity between off-chip access latency and processor core clock frequencies, cache misses are the dominant source of processor stalls for many systems and applications. Inter-processor communication through invalidation-based coherence protocols is the dominant source of cache misses for many shared memory parallel applications. When one processor writes a memory location, all copies of that location must be removed from the caches of other processors. When those processors subsequently access the location, their accesses incur coherence misses. The latency of these misses is exacerbated in multiprocessor systems where cache misses must potentially navigate multiple levels of interconnect before they are serviced. Most current generation out-of-order microarchitectures are designed such that the processor's out-of-order instruction window can tolerate misses that are satisfied from the level-two cache. However, level-two cache misses cause the instruction window to fill and to stall waiting for the miss to return. It is projected that as cache hierarchies grow larger, conflict and capacity misses will be reduced, resulting in an even greater proportion of processor stalls attributed to coherence misses [55].

Figure 6-1 shows the number of misses per 1000 committed instructions for a set of parallel applications running on a four-processor shared memory sys-

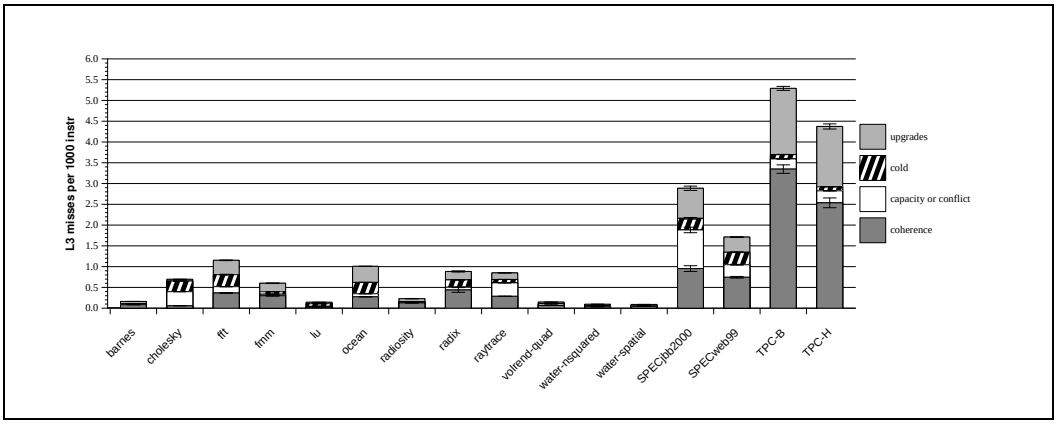

FIGURE 6-1. Misses per thousand instructions for 16MB L3 cache

tem with a directory-based MESI coherence protocol.[1] Each bar is broken into its cold, coherence, and capacity/conflict components [47]. The top of each bar additionally includes upgrade transactions, caused by writes that touch a block for which there is already a local shared copy, which causes inter-processor communication but no data transfer. Many of the applications incur a significant number of coherence misses, especially the four commercial workloads at the right side of the figure. Such misses cause significant performance penalties, particularly in home-based protocols where they must typically make three network hops: from the requester to the home node, from the home node to the current owner, and back to the requester. These three-hop misses average 150 ns latency in current generation directory-based Alpha systems [31]. Coherence misses are also expensive in snooping protocols, especially considering the trend toward hierarchical snooping designs. Although cache-to-cache transfers may cost only 70 ns for misses within a local coherence domain, misses satisfied by external coherence domains can incur directory-like latencies of 160 ns in “snooping” Sun Fireplane systems [23]. Considering the multi-gigahertz clock rates of current microprocessors, these miss

1. Machine configuration details for this data are found in Chapter 7.

latencies account for significant lost instruction execution opportunity.

In an invalidation-based coherence protocol, a block may be invalidated from the cache but the previous copy of the data will remain cache-resident until a subsequent cache miss to that set; the block is marked invalid, but the tag-match logic will indicate a match. The fraction of misses labeled coherence in Figure 6-1 illustrates the frequency of this scenario, in which there is a prior version of the block already cache-resident. In many instances, this data may be useful to the processor. If a cache controller could identify those situations in which it is correct to use the stale data, it could return the stale data non-speculatively rather than stall the processor. This will reduce the latency observed by the processor reading the data, but can also aid the processor that currently has a modified copy of the data. If a new copy is not requested by the reader, then the writing processor maintains an exclusive copy and can continue writing the block without sending an upgrade message.

In this chapter, we describe an implementation of weak ordering called edge-chasing delayed consistency (ECDC). ECDC is a hardware mechanism that identifies stale blocks that can be used non-speculatively, while continuing to provide a coherent and consistent shared memory image to software. Our ECDC implementation improves upon prior versions of delayed consistency by extending the lifetime of stale cache blocks beyond the execution of memory barrier or synchronization instructions by a processor. By detecting cycles in the constraint graph, ECDC allows the use of stale data until a processor becomes causally dependent upon the write that caused the block to become stale.

ECDC is not a new consistency model; it is simply a new way of implementing weak ordering. The principles behind edge-chasing delayed consistency may be applied to the implementation of other consistency models, however due to reasons that will be explained in more detail below, an ECDC implementation of stronger consistency models, such as sequential consistency and processor consistency, will be difficult. However, many commercial architectures utilize relaxed memory models (e.g. PowerPC, IA-64, Alpha), in which an ECDC implementation may improve performance.

We begin our discussion of ECDC in Section 6.1 with a presentation of programming paradigms and microarchitectural artifacts illustrating those scenarios in which it is useful for a processor to continue using stale data after it has been invalidated. In Section 6.2, we describe how the constraint graph representation can be used to identify those cache blocks that can safely be read when stale. In Section 6.3, we present a conceptual description of the edge-chasing delayed consistency protocol, which maintains the constraint graph in a distributed fashion, allowing a processor to use stale blocks from its cache until a cycle has been identified. For clarity, the presentation in Section 6.3 makes no attempt to optimize the protocol in terms of implementation overhead. In Section 6.4, we present an ECDC implementation taking finite resources into account, and discuss the additional hardware structures and modifications that must be made to a conventional shared-memory multiprocessor. We conclude this chapter with a discussion of prior work related to the edge-chasing delayed consistency mechanism in Section 6.5. In Chapter 7, we present a detailed evaluation of the edge-chasing delayed

6.1 Why Delayed Consistency?

It may not be immediately obvious why it would ever be useful to continue using a cache block after it has been invalidated. The programmer updated that data for a reason, right? If she intended to communicate new data from one thread to another, then why would it ever be useful to delay that communication? Of course there are some cases where using stale data will not be useful, even though it may be safe with respect to the consistency model. For example, if a processor acquiring a lock continues to observe the held value of the lock after it has been released, then its acquire will be delayed, reducing performance. However, there are other cases where it does not matter whether the reader observes the old value or the new value; it is more important that the reader reads either of them quickly than wait on the newer value. In the next two subsections, we present examples of applications in which the use of truly shared stale data will improve performance: linked data structures shared among threads, and data-race tolerant iterative convergent algorithms.

Due to false sharing [43] and silent sharing [63], there are also instances in which a block has been invalidated but subsequent loads to that block will read the same value regardless of whether the stale data is returned or the new copy is fetched. This avoidable communication represents another opportunity to benefit from delayed consistency, which we discuss in Section 6.1.3.

6.1.1 Linked data structures

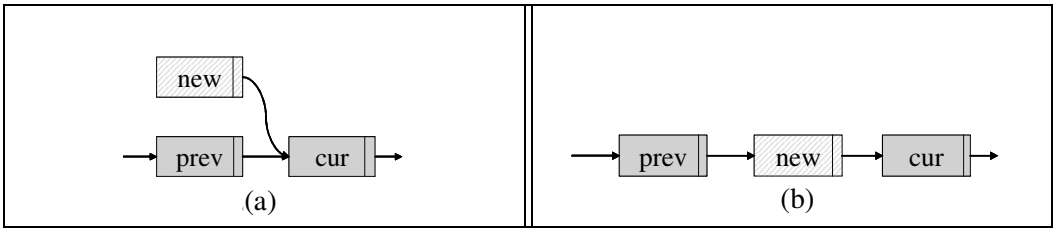

FIGURE 6-2. Lock free list insertion, (a) before CAS, (b) after CAS

Linked data structures are pervasively used in both sequential and parallel applications to create anything from simple linked lists to hash tables to more complex graphs and trees. If a shared data structure is a source of contention in a parallel application, elaborate locking schemes are frequently used to maximize concurrent access by readers and writers. Some algorithms allow readers of a linked data structure to continue to traverse the structure despite the presence of one or more concurrent writers. In such algorithms, a delayed consistency mechanism should provide performance benefit by shielding a processor traversing the data structure from observing (and stalling) to read newly inserted nodes.

For example, Figure 6-2 illustrates a lock-free list insertion occurring in two steps, in which a new node is inserted between two existing nodes. As in any list insertion, the new node's next pointer is first set to the address of the subsequent node. In the second step, a compare & swap (CAS) operation is performed, replacing the old value of *prev*'s next pointer with the new value. If the CAS succeeds, then the new node has been successfully inserted and the operation is complete. A CAS failure indicates that another writer has either deleted *prev* or inserted a new node between *prev* and *cur*, in which case the insertion process must restart. After this insertion occurs, should a reader be traversing this list searching for the node *cur*, or some node after *cur*, there is no reason why that

reader should observe the newly inserted node. If it observes the old version of *prev*, whose next field points to *cur*, then it will simply continue its search at *cur*, without having to stall waiting to read the new node out of a processor's cache across the system.

Lock-free algorithms of this type have been the subject of much recent research [45][76][77][79] and are gaining acceptance at the operating system level with their use in the Linux kernel, IBM/Sequent's Dynix/PTX operating system [75], and IBM's experimental K42 operating system [38]. We present a microbenchmark study of the performance of ECDC running a lock-free list manipulation algorithm of this type in Chapter 7.

6.1.2 Asynchronous communication and convergent iterative algorithms

Delayed consistency should also be useful as a method of implementing asynchronous communication in shared memory multiprocessors. Some applications benefit from the ability to read certain memory locations without caring whether or not the read returns a new version of the data or a previous version. Due to the latency of fetching the data from the most recent writer, the reader can make more forward progress by computing using the old data, rather than stalling while waiting on the new data. Implementing this type of communication is difficult using current instruction sets because they do not support any kind of "don't care" loads. A load reads the newest data, wherever it exists in the system. Using non-binding prefetches is also difficult, because the prefetch must be timed perfectly to return the data before the binding load that subsequently reads the data is executed.

Convergent iterative algorithms are one class of algorithm that use this

model of communication. In parallel versions of such applications, there is typically a shared data structure representing the current state of the solution. Although barrier synchronization may be performed between iterations, the shared copy of the solution is often accessed without synchronization. After a number of iterations, the application converges on a solution. Algorithms of this type include a plethora of parallel equation solvers, sparse matrix factorization (e.g. *cholesky* from SPLASH2 [111]), and many parallel genetic algorithms [106].

6.1.3 False sharing and silent sharing

False sharing, originally defined by Goodman and Woest, is an artifact of the coherence granularity being larger than the smallest addressable unit of memory [43]. A processor p_{writer} may write some portion of a cache block, invalidating that block from another processor p_{reader}'s cache, and cause a miss at p_{reader} even if p_{reader} never subsequently touches the written parts of the block. As will be shown in the next chapter, there is a significant amount of false sharing in some of the workloads studied here, creating communication that we would like to avoid.

Lepak and Lipasti identified another source of unnecessary communication, silent sharing, caused by writes that either overwrite a value with the value already resident at that memory location [65], or revert a location's value to a value that previously existed at that location [66]. Because a read to the stale version of a silently written block will consume the same value that would be consumed if the new data were fetched, delayed consistency can reduce the performance impact of this class of cache misses by using the stale data rather than waiting for the cache miss to return. Lepak found that between 18% and 44% of all

coherence misses were attributable to silent sharing across a set of benchmarks [63].

Delayed consistency protocols can mitigate the performance impact of both false sharing and silent sharing misses. However, not all of these misses will be avoidable. If a processor is already causally dependent upon the write that invalidated a cache block, then it can no longer use the stale block in the ECDC protocol, even if the block is stale due to false sharing or silent sharing.

6.2 Identifying Usable Stale Data

Assuming that we would like to allow a processor to use stale data when possible, how can we identify those blocks for which it is safe? Using the constraint graph, we can identify instances of communication between processors by examining the source and destination processors of each edge. Each $\overrightarrow{dyn-raw}$, $\overrightarrow{dyn-waw}$, and $\overrightarrow{dyn-war}$ edge whose two endpoint instructions were executed by different processors equates to a single miss or upgrade between processors in an invalidation-based coherence protocol. Inter-processor $\overrightarrow{dyn-raw}$ edges correspond to a read miss that is satisfied by a dirty copy of the memory location residing in another processor's cache. Similarly, inter-processor $\overrightarrow{dyn-waw}$ edges correspond to write misses satisfied by remote processors. Interprocessor $\overrightarrow{dyn-war}$ edges correspond to writes that result in either a miss or an upgrade message between processors.

Given a coherence miss caused by a load instruction, we can determine

whether or not that miss is avoidable using the constraint graph based on the following criterion: a $\overrightarrow{dyn-raw}$ edge *e* emanating from writer node *w* and connecting to reader node *r* is necessary if there exists a directed path in the constraint graph from *w* to *r* that does not include edge *e*. This observation follows from Landin's proof that if a constraint graph is acyclic then the execution corresponding to that constraint graph is correct [59]. If *e* is deemed avoidable, then we are essentially transforming the $\overrightarrow{dyn-raw}$ edge from *w* to *r* into a $\overrightarrow{dyn-war}$ edge from *r* to *w*. If there is already a directed path from *w* to *r*, this new $\overrightarrow{dyn-war}$ edge would create a cycle in the graph, and would thus be incorrect. If there is no directed path from *w* to *r*, then the $\overrightarrow{dyn-war}$ edge cannot create a cycle, and the coherence miss is unnecessary.

An illustration of a necessary coherence miss under sequential consistency is shown in Figure 6-3. In this example, processor *p1* is about to perform its second load to cache block A, but the cache block containing A has been invalidated. In order to determine whether or not *p1* can avoid this cache miss, we look at the constraint graph node that would provide the value to the miss (the "W" in RAW), in this case processor *p2*'s store to A. We would like to use the stale value from the cache, thus creating a $\overrightarrow{dyn-war}$ edge from *p1*'s load A to *p2*'s store A (indicated by the dotted arrow). However, if there already exists a directed path from this store node to the load miss node, then we know that this miss is necessary, because the load is already transitively causally dependent upon the store. As we can see in the figure, a directed path already exists from *p2*'s store A to *p1*'s load A through a

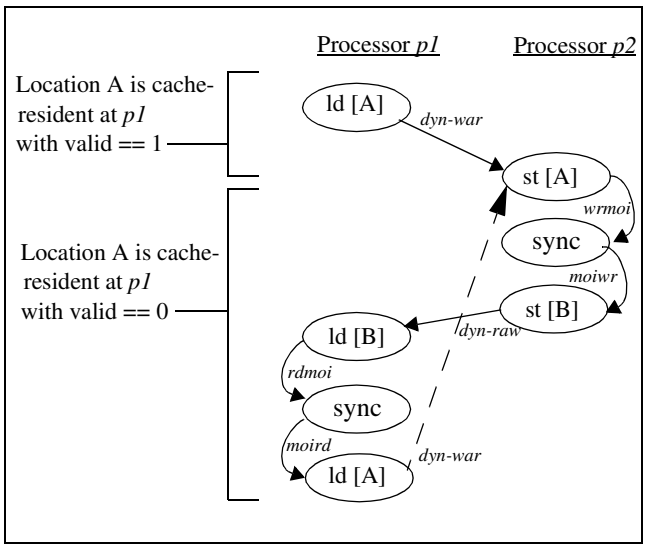

FIGURE 6-3. A necessary coherence miss. (Time progresses from top to bottom.)

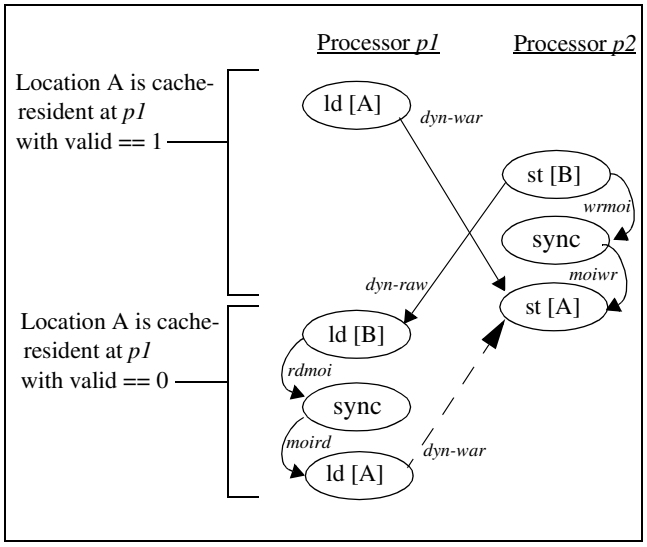

FIGURE 6-4. An unnecessary coherence miss

RAW dependence on memory location B. If this path did not exist, *p1* could safely use the stale value of A. However, because it does exist *p1* must not use the stale value, thus the dotted $\overrightarrow{dyn-war}$ edge must be transformed to a $\overrightarrow{dyn-raw}$ edge, eliminating the cycle.

Figure 6-4 illustrates a similar code segment, however in this case the load miss to location A by *p1* is avoidable. The miss is avoidable because we can create a legal schedule of operations such that the load miss will return the old value of A.

In this example, *p2* performs the stores to memory location A and B in reverse order. Consequently, at the time of *p1*'s load miss, there is not already a directed path from *p2*'s store of A to *p1*'s load of A, therefore the $\overrightarrow{dyn-war}$ edge caused by using the stale value does not create a cycle, and this coherence miss is avoidable.

Speculative mechanisms might guess that it is safe for *p1* to use the stale data in A because the store to A by *p2* may have been a silent store or may have been to a different word within the same cache line. These mechanisms require a verification step to ensure that *p1* did indeed load the correct value. Given the long latencies associated with fetching data from another processor's cache, this verification operation will most likely stall the processor. However, using the constraint graph, we can detect cases where it is safe to use the stale value, without the need for verification.

6.3 Edge-Chasing Delayed Consistency: A Conceptual Description

In this subsection, we describe the concepts behind an implementation of the weak ordering memory model called edge-chasing delayed consistency. In order to provide a clear description of this new caching algorithm, we separate the idea from the implementation and present a purely theoretical description in this subsection, not subject to any hardware constraints. In Section 6.4, we describe a hardware implementation of the algorithm, which we experimentally evaluate in Chapter 7.

Edge-chasing delayed consistency derives its name from a class of dead-

lock detection algorithms proposed for distributed database systems [22][54]. Processes in such systems can optimistically acquire and release locks as desired (i.e. do not follow rigid deadlock-free locking disciplines), and in the event of deadlock, abort one of the transactions participating in the deadlock, thus freeing that transaction's held locks and allowing the other transactions to proceed. Many algorithms perform detection through the construction of a *waits-for-graph* (WFG), a directed graph whose nodes correspond to processes, and whose edges represent the dependences among processes (specifying which processes "wait-for" which other processes). For example, if a process A is blocked attempting to acquire a lock held by process B, there will be an edge from A to B in the WFG. A deadlock can be detected by testing the WFG for a cycle; if a cycle exists, then there must be a cyclic dependence of resources held by processes.

Edge-chasing algorithms detect cycles in the WFG in a distributed fashion through the propagation of special messages called *probes* communicated along the edges of the graph. When a process suspects the existence of a deadlock due to a timeout, it creates a probe and sends the probe to the process on which it waits. The recipient of the probe forwards the message on to the process on which it waits. The reception of a probe created by the receiving process indicates that the process is part of a cycle in the graph, causing the process to abort.

The problem of maintaining consistency in a shared-memory multiprocessor resembles the deadlock detection problem. Instead of checking the WFG for cycles, we will check the constraint graph for cycles using a similar mechanism. Keep in mind that in neither the deadlock detection scenario nor the memory con-

sistency scenario do we need to explicitly construct or communicate the entire graph. Instead, the occurrence of a cycle can be inferred by the receipt of a locally created probe.

At a high level, edge-chasing delayed consistency works as follows: every write operation that invalidates a cache block from a remote cache initiates the creation of a probe.[1] A probe is a globally unique identifier that is passed from processor to processor at the occurrence of certain events. A copy of this probe is kept with the stale copy of the invalidated cache block. The stale block may be used until the block's associated probe is received from another source.

When communicating with other processors through loads and stores to shared memory, the creator of a probe ensures that stale copies are not used incorrectly by passing the probe on to the other processors whenever the other processor's load or store will follow the probe-initiating write in the constraint graph. For example, after invalidating a remote cache block *c* and creating probe *pb*, processor *pwriter* writes a different memory location that is subsequently read by another processor *preader*. When *pwriter* sends the new data to *preader*, it also sends probe *pb*, because *preader* is now causally dependent upon processor *pwriter*'s write, meaning that if *preader* subsequently reads cache block *c*, it should observe the new copy of the block. By passing probes only along edges in the constraint graph, edge-chasing delayed consistency ensures stale data will remain useful as

1. Unfortunately, the term "probe" is sometimes used as a synonym for invalidation messages in coherence protocols. In this thesis, we exclusively use the term "invalidation" to refer to write messages such as the *upgrade* and *get exclusive* transactions found in coherence protocols, and exclusively use the term probe to refer to the additional identifiers used in edge-chasing algorithms.

long as possible: until the processor using the data becomes causally dependent upon a newer version of the stale block.

In order to support this communication, edge-chasing delayed consistency maintains sets of probes for each processor and memory location, indicating the writes on which they causally depend. When a coherence message is sent, the probe set corresponding to that memory location is sometimes attached to an outgoing message. When a coherence message is received, the probe set attached to the incoming message is added to a per-processor probe set and the memory location's probe set. A more precise description follows. We present a simple example of ECDC operation in Section 6.3.2.1, which should clarify the purpose of various choices specified in the formal description.

6.3.1 Formal description

We assume a weakly ordered (PowerPC) system model consisting of a set of (in-order issue) processors $P = \{ p_1 \ldots p_n \}$, where each processor can execute load and store operations to an associated cache $c_1 \ldots c_n$, and can also execute memory barrier instructions that order these operations[1]. To simplify our discussion, we assume each cache has unlimited capacity and initially contains the contents of the entire memory. Instructions are ordered with respect to one another as described in Chapter 2 for weak ordering. Each cache block can be in one of three states corresponding to the three states in a standard MSI invalidation-based coherence protocol. A load access reading a block at address a that is in the invalid state

1. Assuming in-order issue processors is not a fundamental limitation, but simplifies this discussion. When discussing our ECDC implementation in the next section, we describe the minor support needed for out-of-order microprocessors.

initiates a broadcast *ReadRequest(a)* message, which results in a *ReadResponse(a, d)* message containing data *d* from other caches that contain a valid copy of the block. A store operation writing a block at address *a* that is in the shared or invalid state initiates a broadcast *WriteRequest(a)*, which results in a *WriteResponse*(*a, d*) message containing data *d* from those other caches in the system that contain a valid copy of the block.

Edge-chasing delayed consistency makes the following additions to this system model:

- **Extra Stable State:** The MSI states are augmented with an additional state, the stale (ST) state. Loads to the stale state may be satisfied locally, and stores to the stale state are handled identically to stores to the invalid state.
- **Supplanter Probes:** For each cache $cache_i$ and cache block *a* in the stale state in $cache_i$, a probe $\phi_{supplanter,i,a}$ is maintained. This probe is a copy of the probe corresponding to the write that invalidated *a*, and it is used to determine when *a* can no longer be read. (The term *supplanter* is used because the stale version of the cache block has been supplanted by the version corresponding to this probe's write).
- **Processor Upstream Set:** Each processor $proc_i$ maintains a set of probes $\Phi_{procupstream,i}$ called its *upstream set*. The upstream set contains those probes on which the processor is currently causally dependent (that will be "upstream" in the constraint graph from instructions subsequently executed by the processor).

- **Per Location Read and Write Upstream Sets:** For each cache $cache_i$, each cache block *a* also has two associated upstream sets, a *read upstream set* $\Phi_{readupstream,i,a}$ and *write upstream set* $\Phi_{writeupstream,i,a}$. These sets contain those probes that a sub sequent reader, or writer, respectively, will be causally dependent upon after reading or writing cache block *a*.
- **Probes in Messages:** In addition to the usual fields included in each message sent between processors, response messages carry a probe set, i.e. *ReadResponse(a, d,* Φ*),* and *WriteResponse*(*a, d,* Φ*).* Write requests additionally carry a single probe, i.e. *WriteRequest(a,* ϕ*)*. We also add a new message that carries two probe sets *PropePropagate(*Φ_{keys}, Φ_{new}*)*, which is sometimes broadcast at memory barrier executions in order to convey new causal dependences that a supplanting write should be dependent upon if the corresponding stale block is subsequently read. Upon receiving a ProbePropagate message, processors respond by sending a *ProbePropagateResponse*() message, which carries no data or probes.

Initially, for each cache block *a* and processor *i*, the $\phi_{supplanter,i,a}$, $\Phi_{readupstream,i,a}$, $\Phi_{writeupstream,i,a}$ and $\Phi_{procupstream,i}$ structures are set to null. Table 6-1 specifies invariants concerning the contents of each probe set, which are maintained using the operations specified in Table 6-2. Table 6-2 only specifies the protocol operation for those transitions where the protocol differs from a typical MSI protocol. The upper part of the table specifies the actions that are taken when handling load and store operations, the first four corresponding to miss events and the next two corresponding to hit events. The lower part of the table specifies the

Table 6-1: The invariants of probe sets

Probe Set	Invariant Definition
Per processor upstream set	For each miss-causing write w on which processor p_i is causally dependent, there exists a probe ϕ_w that is a member of p_i's upstream set $\Phi_{procupstream,i}$
Per location read set	After a memory location a has been written by a processor p_i, the read set $\Phi_{readupstream,i,a}$ contains the union of p_i's per processor upstream set and the previous contents of $\Phi_{readupstream,i,a}$.
Per location write set	After a memory location a has been read or written by a processor p_i, the write set $\Phi_{writeupstream,i,a}$ contains the union of p_i's per processor upstream set and the previous contents of $\Phi_{writeupstream,i,a}$. Consequently $\Phi_{readupstream,i,a}$ is a subset of $\Phi_{writeupstream,i,a}$.

actions taken at memory barrier operations. The event that first sets the ECDC protocol in motion (labeled 1.) is a store miss that invalidates a cache block from one or more other processors' caches. When the store miss occurs, the processor creates a probe and attaches it to an outgoing broadcast *WriteRequest* message. The probe is also added to the processor's upstream set.

If the receiver of the *WriteRequest* message (event 2) has a valid copy of the cache block, the receiver sets the block's state to ST, and sets the block's $\phi_{supplanter}$ to the incoming probe. The receiver responds with a *WriteResponse* message containing a copy of the data and the write upstream set for the requested cache block. The cache block's write upstream set contains those probes on which any subsequent writer to the cache block will causally depend., including $proc_i$'s store.

Upon the reception of the *WriteResponse* message (event 3), the writing processor inspects the probe set attached to the message. If the probe set contains any probes that correspond to the $\phi_{supplanter}$ probe for any of its cache's stale blocks, that block's state is set to invalid; the reception of this probe indicates that

Table 6-2: Edge-chasing delayed consistency state transition table. Load/store operation in upper section, memory barrier operation in lower section.

Event	Action(s)
(1) Store by $proc_i$ to block b in state I or S	Create probe ϕ. Send *WriteRequest(a,* ϕ *)* to all processors. Add ϕ to $\Phi_{procupstream,i}$.
(2) Reception of *WriteRequest(a,* ϕ *)*, by $proc_j$ from $proc_i$, to block a in state M or S	Set state of cache block to ST, set $\phi_{supplanter,j,a}$ to ϕ. Send *WriteResponse(a, data,* $\Phi_{writeupstream,j,a}$*)* to $proc_i$.
(3) Reception of *WriteResponse(a, data,* Φ *),* or *ReadResponse(a, data,* Φ *)*, by $proc_i$ from $proc_j$	For each probe ϕ in Φ*:* For all blocks b in $cache_i$ in state ST: if $\phi = \phi_{supplanter,i,b}$, set block b's state to I. Add ϕ to $\Phi_{procupstream,i}$. Add ϕ to $\Phi_{writeupstream,i,a}$. If message was *WriteResponse*, also add ϕ to $\Phi_{readupstream,i,a}$.
(4) Reception of *ReadRequest(a)*, by $proc_j$ from $proc_i$, to block a in state M or S	Send *ReadResponse(a, data,* $\Phi_{readupstream,j,a}$*)* to $proc_i$.
(5) Store by $proc_i$ to block a in state M	For each probe ϕ in $\Phi_{procupstream,i}$: Add ϕ to $\Phi_{writeupstream,i,a}$. Add ϕ to $\Phi_{readupstream,i,a}$.
(6) Load by $proc_i$ to block a in state M, S	For each probe ϕ in $\Phi_{procupstream,i}$: add ϕ to $\Phi_{writeupstream,i,a}$.
(7) memory barrier executed by $proc_i$	Broadcast *ProbePropagate(*Φ_{keys}, $\Phi_{procupstream,i}$*)*, where Φ_{keys} contains $\phi_{supplanter,i,b}$ for all blocks b in the ST state at $cache_i$. Disable stale data use.
(8) Reception of *ProbePropagate(*Φ_{keys}, Φ_{new}*)* by $proc_i$ from $proc_j$	For each probe ϕ_{key} in Φ_{keys}: if $\Phi_{procupstream,i}$ contains ϕ_{key} For each probe ϕ_{new} in Φ_{new}: Add ϕ_{new} to $\Phi_{procupstream,i}$. For all blocks b in $cache_i$ in state ST: if $\phi_{new} = \phi_{supplanter,i,b}$, set block b's state to I. Send *PropePropagateResponse()* to $proc_j$
(9) Reception of *ProbePropagateResponse*() from all processors	Enable stale data use.

the processor is now causally dependent upon the write that originally invalidated the stale cache block, and therefore the stale data should no longer be used. The incoming probe set is also added to the processor's upstream set, and the read and write upstream sets for that particular cache block. It is added to both sets because both subsequent reads and subsequent writes to the block will be causally dependent upon this write.

Read requests in the ECDC protocol occur identically to reads in a conventional coherence protocol, except responses include a copy of the read upstream set for that block in addition to data. The read upstream set contains those probes on which a reader to the cache block will causally depend. Just like the handling of the *WriteResponse* message, when the *ReadResponse* message returns, this probe set is used to transition stale cache blocks to the invalid state if the reading processor now causally depends on the write that forced the cache block to become stale. Also like the handling of a *WriteResponse* message, we add the incoming probe set to the upstream set of this processor, and to the write upstream set of this memory location (the incoming probe set is not added to the block's read upstream set because subsequent reads by other processors will not be causally dependent upon this read, i.e. no RAR dependences).

Cache hits (events 5 and 6 from Table 6-2) update the contents of the referenced cache block's upstream set (write upstream set in the case of loads, both read upstream set and write upstream set in the case of stores) by adding all of the probes in the processor's upstream set to the block's appropriate upstream set, thus ensuring that those probes on which the processor currently causally depends will be passed to processors that may read or write the memory location in the future.

After a block enters the ST state, a processor may still become dependent on a new probe. Should the processor subsequently use that block while in the ST state, the original supplanting writer will also need to become causally dependent upon this new probe. Propagating these extra dependences after a block's initial invalidation are the purpose of the events labeled 7 through 9 in Table 6-2. When a

processor executes a memory barrier, it sometimes broadcasts a *ProbePropagate* message to the other processors in the system with one probe set containing the supplanting probes for each of the blocks in the ST state (the "keys" set), and a second probe set corresponding to the processor's upstream set (the "new" set).[1] The processor then temporarily disables the use of stale data until a *ProbePropagateResponse* message has been received from all of the other processors. The *ProbePropagate* message does not need to be sent at every memory barrier operation, because sometimes either the keys probe set or the new probe set in the outgoing message will be empty, eliminating the need to send it.

When a processor receives a *ProbePropagate* message, it checks to see if its upstream set contains any of the probes in the incoming message's keys probe set. If so, this indicates that this processor is already causally dependent upon that probe, so the processor should also become causally dependent upon the new probes in the message. As in the other cases, when becoming causally dependent upon new probes, any blocks in the ST state whose supplanting probe corresponds to the new probe are forced to transition from ST to I.

This concludes our theoretical description of the edge-chasing delayed consistency protocol. Before describing a feasible hardware implementation, we present two short examples illustrating the operation of the protocol.

6.3.2 Examples of operation

1. A broadcast operation is a sufficient mechanism for correctly propagating probes. In large systems, it may be advantageous to send ProbePropagate messages to the subset of processors that are causally dependent upon the supplanting write, rather than the entire system.

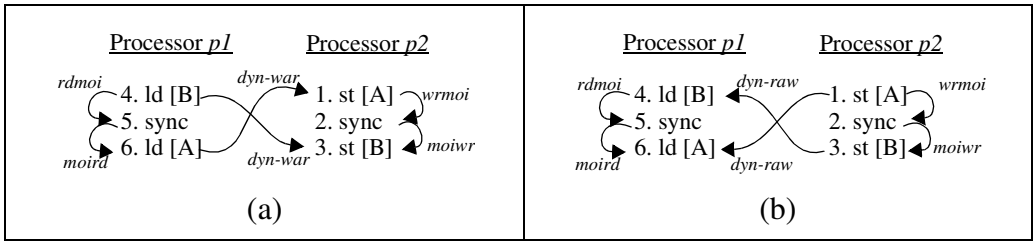

FIGURE 6-5. Simple ECDC example:(a) Initially, p1's cache contains blocks A and B in the M state. (b) Initially, p1's cache contains block A in the M state, but block B is invalid.

In this subsection, we present a set of examples illustrating the operation of the theoretical edge-chasing delayed consistency protocol. We begin with a simple example in which the communication of probes by *ProbePropagation* messages sent at memory barrier instructions is unneeded. The subsequent example described in Section 6.3.2.2 illustrates the use of these messages.

6.3.2.1 A simple example

Figure 6-5 shows two different executions that can occur for a two-processor code segment, even though the instructions are executed in an identical order in both examples (the execution order is labeled next to each instruction). Depending on the initial state of blocks A and B in processor *p1*'s cache, the ECDC protocol may or may not allow *p1* to use stale copies of either block. In Figure 6-5(a), *p1* is able to use stale copies of the cache blocks. This is not the case for the example shown in Figure 6-5(b), resulting in a different outcome for the execution (also resulting in two different constraint graphs).

For the example shown in Figure 6-5 (a), let's assume that both block A and block B are initially in the M state at *p1*. Processor *p2*'s store to block A (operation 1) will be a cache miss, resulting in the creation of a new probe and the initiation of a *WriteRequest* message sent from *p2* to *p1* containing this probe. Processor *p2* also adds the probe to its upstream set. If we assume that *p2* does not

currently have any blocks in state ST, when it executes its *sync* operation, it does not need to broadcast a *ProbePropagate* message, because the probe set Φ_{keys} in the outgoing message would be empty. When *p2* executes its store to block B (operation 3), it performs the same actions it performed when executing its first store: create and send a probe with its invalidation message and add the probe to its upstream set.

Since processor *p1*'s cache contained blocks A and B in the M state (this example would be identical if they were in the S state) before *p2*'s writes, when it receives each of *p2*'s invalidation messages, it will transition the state of each block to the ST state, and keep a copy of the incoming probe in the block's supplanting probe field $\phi_{supplanter}$. When *p1* subsequently performs its load operations (operations 4 and 6), because it has not received *p2*'s probes via any incoming messages, the blocks will still be in the ST state, and *p1*'s loads can read them and retire without fetching the new data from *p2*. In between the two load operations, *p1* executes a sync operation. If we assume that *p1* has not added any probes to its upstream set since its last sync operation, then it will not need to broadcast a ProbePropagate message either, because the outgoing message's set of new probes (Φ_{new}) will be empty.

Given this initial cache state, processor *p1* is able to use stale versions of both cache blocks, saving it from stalling while fetching the newest copy of the data. The outcome is different if one of the cache blocks is not initially cache resident at *p1*. Starting this example over, lets assume that *p1*'s cache initially contains block A in the M state, but block B is invalid. Shown in Figure 6-5 (b), the exam-

ple would begin similarly: when *p2* performs its store to block A, it would send an invalidate containing an associated probe ϕ to *p1*, and *p1* will associate ϕ with the stale copy of A by setting A's supplanter field. However, when *p2* performs its store to block B, because *p1* does not have a valid copy of B, *p1* will not necessarily observe the invalidate. When *p1* performs its load to block B, it will fetch the new copy from *p2*. Processor *p2* will respond to *p1* with a *ReadResponse* message containing the contents of block B's read upstream set (including probe ϕ). When *p2* receives the response, it will check to see if there are any blocks in the ST state with a supplanter field set to ϕ. In this case, it will find block A, and set its state to I. Consequently, when *p1* performs its load to A, it will be forced to fetch the new copy from *p2* instead of using a stale copy, thus preventing a cycle from forming in the constraint graph.

6.3.2.2 Dekker's algorithm

Our next example is similar to the first, except it illustrates two processors that attempt to ignore each other's writes, as opposed to the previous single-writer example. For the execution in Figure 6-6, initially, processor *p1*'s cache contains a modified copy of block A, and *p2*'s cache contains a modified copy block B. The execution begins with a write miss to block B by p1 and a write miss to block A by *p2*. When each of these write misses is initiated, a probe is created by each processor, and sent along with the *WriteRequest* message. At the reception of this message, each processor transitions the appropriate cache block to the ST state, copies the incoming probe to the block's $\phi_{supplanter}$ field, and responds with a *WriteResponse* message containing a copy of the block's $\Phi_{writeupstream}$ probe set (which is

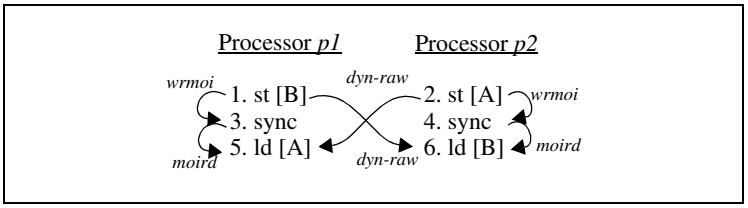

FIGURE 6-6. Dekker's algorithm. Initially, p1's cache contains blocks A and B in the M state.

currently empty).

When each processor executes its *sync* instruction, it broadcasts a *ProbePropagate* message to the other processors in the system. This message includes a *keys* probe set containing the supplanting probes for each of the blocks in the ST state (for *p1*'s message, this will contain the probe for block B, for *p2*'s message, the probe for block A), and a *new* probe set (For *p1* this will contain the probe for block A, and for *p2*, this will contain the probe for block B). After sending the *ProbePropagate* message (and waiting for its writes to be ordered, as usually happens at memory barrier instructions), each processor can continue executing instructions. However, the processor cannot use any stale data until receiving a *ProbePropagateResponse* message from the other processors.

In this example, each processor will receive a *ProbePropagate* message, with a keys set containing the probe for the block that it just invalidated. Processor *p1* receives *p2*'s *ProbePropagate* message with keys set containing the probe for *p1*'s initial write to block B. This block is part of *p1*'s upstream set, indicating that *p1* is already causally dependent upon the write, which means that *p1* should also become causally dependent upon the probes in the incoming message's new set.[1] Once a processor becomes causally dependent on a new probe, it must throw out any stale blocks whose supplanter field matches the new probe, in this case block

A. The arrival of *p1*'s *ProbePropagate* message at *p2* causes a similar sequence of events, resulting in the transition of block B from state ST to state I in *p2*'s cache.

After each of these blocks transition to the I state, the processors perform their loads, which incur a cache miss to fetch the new data, and result in the constraint graph shown in Figure 6-6. Of course, depending on timing, it is also possible that one of the loads read the stale data if the load happens to execute before the arrival of the other processor's *ProbePropagate* message. However, because neither processor can use stale data until its *ProbePropagate* message has been acknowledged by the other processors in the system, it is not possible for both processors to use the stale data, thus preventing an incorrect execution of Dekker's algorithm.

This example can also be used to explain why an implementation of edge-chasing delayed consistency that implements stricter models will be difficult. Under sequential consistency (and some other strict models), if each of the loads returns the stale value, it would result in an inconsistent execution, regardless of the presence of memory barriers separating the stores and loads. Consequently, an ECDC implementation would need to propagate probe information every time a processor becomes causally dependent on a new probe, rather than delaying this propagation until the next memory barrier. These extra probe propagations are necessary to ensure that once a processor becomes causally dependent upon a new

1. We preemptively force *p1* to become causally dependent upon these probes under the assumption that *p2* will use the blocks that are in its cache in the ST state. This assumption is conservative. For example, p2 keeps a stale copy of B, but if we assume operation 6 doesn't exist, p2 may never use the stale copy. If this is the case, then it is safe for p1 to use the stale version of A. However, in the absence of an oracle mechanism that can predict whether a stale block will be used in the future, the ECDC protocol must conservatively disallow use of the stale blocks.

probe, this new probe will be passed through any WAR edges that will be created when that processor uses stale data in the future.

6.4 Mapping Edge-Chasing Delayed Consistency to Hardware

In this section, we describe a hardware implementation of the edge-chasing delayed consistency protocol. Because ECDC communicates probe information as part of response messages to write requests (in addition to other messages), it is best suited to a multiprocessor design utilizing a home-based coherence protocol which already requires invalidate acknowledgment messages. Such protocols, although originally intended for large scale multiprocessing [60][62][102], have been gaining traction recently in small-to-medium scale multiprocessors [3][15][41] due to their advantages in interconnect flexibility (no need for ordered interconnect).

Our ECDC implementation is based on an SGI-Origin-like MESI directory-based coherence protocol. Figure 6-7 illustrates one processing node in such a system, including shaded boxes highlighting the new or modified components needed for implementing the ECDC protocol. Our implementation requires the addition of three main components: the stale address buffer (STAB), the probe propagation buffer (PPB), and the cast-out PPB. A single STAB and PPB is associated with each processor in the system, and a cast-out PPB with each directory/memory controller. Additionally, each processor's data caches are slightly modified, with one extra stable state (the ST state), and one extra bit per cache block (used to identify potential synchronization locations). We make no modifi-

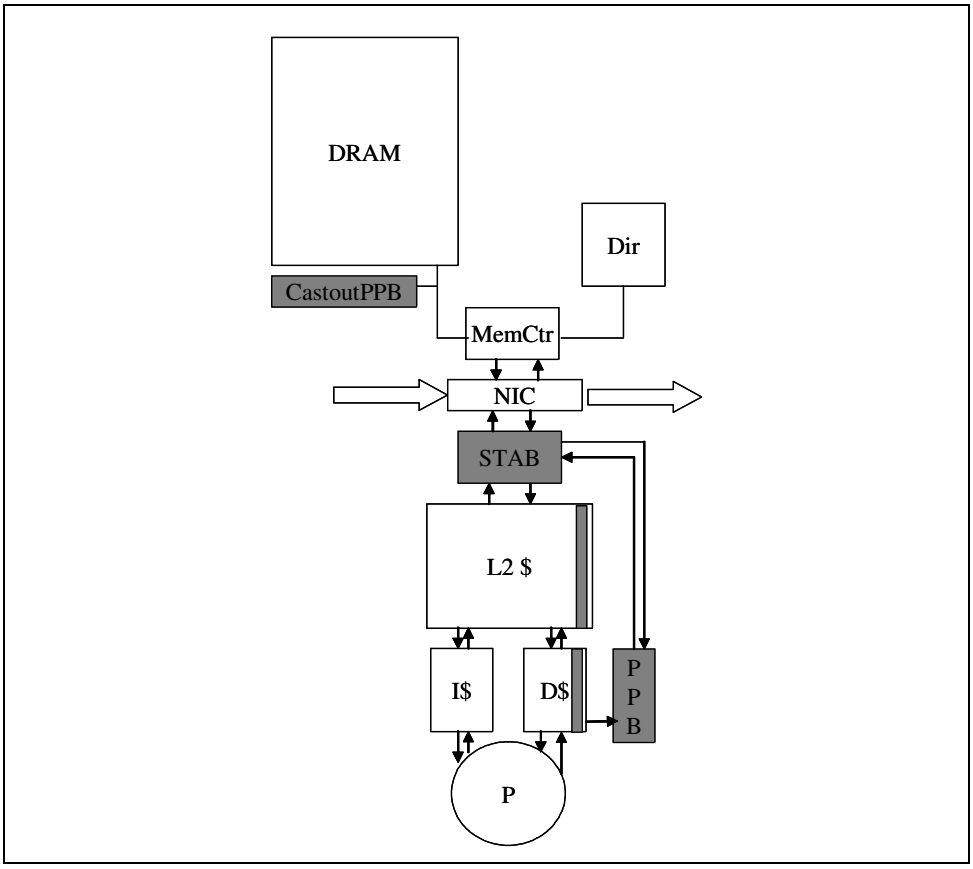

FIGURE 6-7. ECDC system modifications. (Not drawn to scale.)

cations to the processor core itself. At a high level, the purpose of the STAB is to maintain the $\phi_{supplanter,a}$ set for each stale cache block *a,* monitoring incoming probe sets and signaling the cache to invalidate the stale block appropriately. The PPB is used to maintain a superset approximation of the read upstream set $\Phi_{readupstream,i,a}$ and write upstream set $\Phi_{writeupstream,i,a}$ for each block recently touched by the processor. Certain outgoing coherence response messages are augmented with an upstream probe set determined by the PPB. To avoid the loss of probe information when writing back a cache block to memory (including write backs caused by coherence messages), we also include a simplified version of the PPB at the memory/directory controller, which we call the CastoutPPB.

This section is organized as follows: we first describe the representation

that we use for probes and probe sets, a choice that affects all other aspects of the protocol. We then describe the ECDC-related components highlighted in the figure. We conclude with a few examples of ECDC operation for typical protocol events, and a discussion of other miscellaneous effects of the ECDC implementation.

6.4.1 The representation of probes and probe sets

The choice of probe representation is the most important aspect of this design, affecting the complexity of every other part of the implementation. Logically, a probe is a globally unique identifier, and a probe set is collection of such identifiers. The representation choice is based on a trade-off between allowing the most opportunity for using stale cache blocks, by building a system functionally equivalent to the theoretical ECDC description from Section 6.3, while optimizing the protocol to be feasible in terms of the hardware's physical area and complexity. A good probe representation should have the following properties:

- **space-efficient:** it should be frugal in terms of hardware resource consumption, and allow space efficient probe set implementations that can contain an arbitrary number of probes.
- **reusable names:** because the probe namespace is finite, there must be a simple way to reuse a probe name after every processor in the system is causally dependent upon the write that created that probe (rendering the associated stale data and probe useless).
- **precise representation of causality:** If the first two requirements are optimized too aggressively, causal information may be lost, resulting in a con-

servative approximation of causality that reduces the opportunity to be gained from using this technique. An ideal representation will be efficient in terms of area and complexity, while also precisely maintaining the causal relationships among processors.

The edge-chasing delayed consistency protocol has been through several revisions, each simplifying the probe representation while attempting to accurately reflect causality. Due to the importance of the representation itself, in this section we describe the evolution of our choice, giving insight into its design rather than simply stating “here it is”.

In the original version, probes are identified by a unique <*processor id, integer probe id*> pair, and each processor maintains its own namespace by assigning probe identifier integers arbitrarily based on availability. Once allocated, probe identifiers are reused only after there are no remaining stale copies of the corresponding block in the system. Determining this termination condition requires a probe garbage collection protocol, adding significant complexity and increasing the bandwidth demands of the system. Furthermore, probe sets become quite large, limiting the feasibility of the implementation.

The representation’s second version eliminates the garbage collection problem by augmenting probes with a timeout value. Timeout values dictate the expiration of stale cache blocks (in the $\phi_{supplanter}$ field), the expiration of a probe from a probe set, and the time at which the processor that created the probe can reuse the name of the probe. The timeout associated with each probe is initialized with a global constant that we refer to as the *stale block lifetime*. The stale block

lifetime parameter should be set in such a way that it is long enough to capture any opportunity to use stale data, but not so long that the associated probe is alive long after the stale block can no longer be used, consuming excess space. (During our performance evaluation in Chapter 7, we explore various settings for this important parameter.) The probe timeout is maintained as a counter that is decremented every *n* processor clock cycles. We assume this clock is based on a loosely synchronous clock, similar to the global clock used for the construction of checkpoint numbers in Sorin et al.'s SafetyNet implementation, whose skew is less than the message communication time between any two nodes [104]. The parameter *n*, which we call the *decrementer constant*, affects the trade-off between probe storage and timer resolution. For the data collected in this thesis, we assume a decrementer constant of one, however larger decrementer constants may be used in practice in order to save space consumed by counters or to ease the maintenance of counters (should decrementing every counter every cycle prove difficult). In Chapter 7, we present an analysis of how the size of various structures would be affected by larger decrementer constants.

The second version also improves upon the first by reducing probe set size by assigning integer probe names using consecutive integers reflecting the single-thread ordering of the writes that create the probes. For example, if a processor has two store misses, the probe identifiers corresponding to those stores will be assigned such that the earlier store in program order will have a smaller probe identifier. If those misses were initiated out of program order (e.g. due to a non-binding exclusive prefetch), the identifier for the earlier instruction (earlier in pro-

gram order, subsequent in order of miss initiation) is assigned a probe identifier less than the outstanding miss probe identifier.

Because store misses initiated by each single processor are ordered by probe identifier, a probe set in an *n*-processor system must contain no more than *n* probes; any more would be redundant. The probes are redundant because if a processor or memory location is causally dependent upon the newer of two writes by a single processor, it must also be dependent upon the earlier write. In this representation, probe identifiers become the triplet <*processor id, integer probe id, integer timeout value*>. Probe sets become arrays of these triplets, with one entry per processor in the system, resembling the vector clock representation that has been widely used in the context of distributed systems [68][88][105]. When adding a new probe to a probe set, the probe set entry corresponding to the new probe is overwritten only if the new probe identifier is greater than the existing identifier.

The third (and final) representation is based on the second version, except its size is reduced by discarding the integer probe identifier. Rather than detecting the existence of a cycle in the constraint graph by comparing the integer probe identifiers, we can detect a cycle by comparing timeout values. If any supplanter probe timeout value is less than or equal to the corresponding entry in an incoming message's upstream set, we know that the supplanting write is causally dependent on the incoming data, and must throw away the stale data. In this representation, when probes are stored individually (i.e. as the $\phi_{supplanter}$ field for a memory location), they are stored as the tuple <*processor id, integer timeout value*>. To represent probe sets in systems with a small number of processors, the processor id field

can be discarded, because it can be inferred by the position of the timeout value within the probe set vector. When used in systems with a large number of processors (resulting in sparse probe sets that commonly only contain probes created by many fewer processors than the number of processors in the system), it will be advantageous to explicitly maintain the processor id, and condense the vector so that it contains tuples for only those processors with active probes. We use the representation tailored to small systems for this thesis.

To summarize, probes are represented by a <*processor id, timeout*> tuple. Probe sets are represented by a vector of timeouts, with the *procid* corresponding to each timeout being positionally inferred. Maintaining probe timeout values in the ECDC protocol is not as difficult as in other distributed timestamping network protocols [70][94], because the ECDC protocol does not require as much precision from its timers. So long as $\phi_{supplanter}$ probes do not expire late, and probes that are part of an upstream set do not expire early, the protocol will operate correctly. Both timers can be approximated as long as the approximations skew the time in the correct direction. This flexibility eases some of the implementation burden associated with timestamp maintenance. For example, we can avoid decrementing timeout values for in-flight probes at each network switch. Instead when the receiving processor receives an invalidate and associated probe, it can set the $\phi_{supplanter}$ field for the stale block by subtracting a maximum network transit time from the stale block lifetime parameter. Consequently the probe identifier does not even need to be included in the invalidate message; instead, the identifier can be constructed at the destination. At worst, the invalidate has arrived much earlier than

the maximum transit time, unnecessarily reducing the lifetime of the stale cache block (a performance divot, but not a correctness problem). Similar optimizations exist for storage structures containing probe sets (described in detail in the next few subsections); if it is difficult to decrement the timer value every cycle for every probe in a probe set, timer decrements can be delayed, causing the probe to expire later (consuming more space, but not causing a correctness problem).

One drawback of the timer-based representation is that it is not possible to exactly maintain a weakly ordered constraint graph using a single integer per processor; instead, a conservative approximation is maintained. As shown in Figure 2-4 on page 20, there may be multiple strands of ordered instructions between two memory barriers. The use of a single integer flattens these strands into a single strand, ordered by the time at which each store miss occurs. Consequently, the timer-based scheme may order some stores that do not necessarily need to be ordered (e.g. from the figure, unnecessarily ordering one of the stores to A with respect to one of the stores to C). Because there can be arbitrarily many independent strands of execution between memory barriers (subject to the limits of the machine's address space), there is no feasible hardware representation that can exactly describe this constraint graph. We believe that the compactness provided by the timer-based representation outweighs the benefits that could be obtained by more precisely ordering a processor's writes that occur between two memory barriers.

6.4.2 Stale address buffer (STAB)

The *stale address buffer* (STAB) is used to control the transition of cache

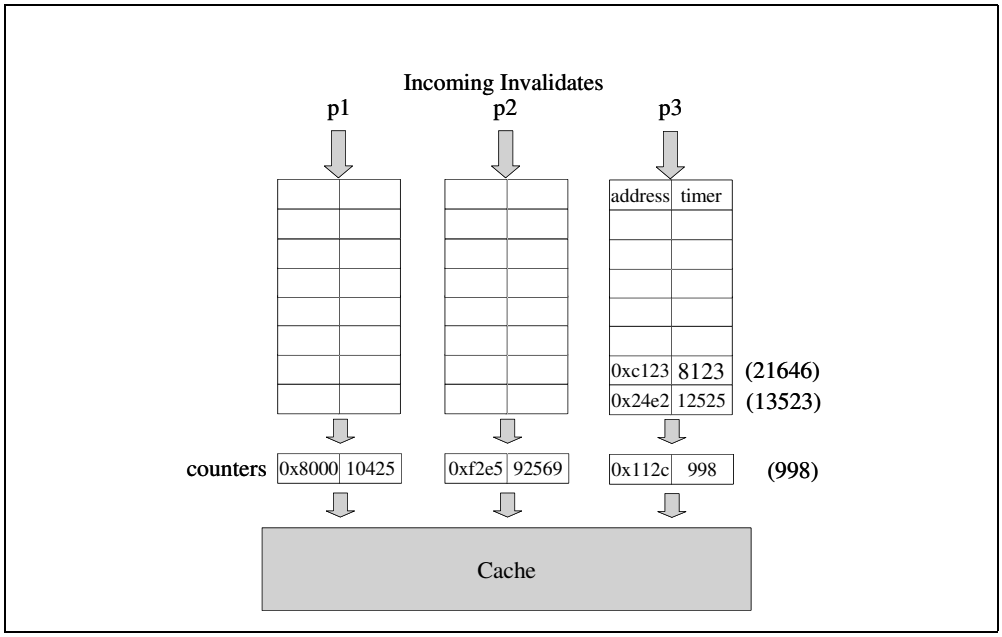

FIGURE 6-8. STAB design for a 4-processor system

blocks from state ST to state I, preventing the processor from reading stale data when doing so would be incorrect. This job is performed by maintaining the $\phi_{supplanter}$ probe for each cache block that is currently in the stale state. Functionally, probe maintenance involves two operations: 1) decrementing the timer values of each $\phi_{supplanter}$ probe; and 2) monitoring the probe sets attached to incoming messages, removing $\phi_{supplanter}$ probes when a probe originating from the same processor is observed whose timeout is greater than the $\phi_{supplanter}$ probe's timeout. Once a $\phi_{supplanter}$ probe expires for either of these reasons, a signal is sent to the cache hierarchy, which transitions the block's state from ST to I.

The STAB consists of a set of FIFO queues, one per other processor in the system. Figure 6-8 illustrates a STAB logical design in the context of a four-processor system. Each FIFO entry contains the $\phi_{supplanter}$ probe and block address for an incoming invalidation. At the arrival of an invalidation message, the invali-

dation is sent to both the cache hierarchy and the STAB. If the block is valid in the cache hierarchy, the block's state transitions to the ST state, and the STAB is notified that the block has entered the ST state, causing the STAB to store the incoming invalidate's address and probe in the appropriate FIFO queue (whose tail is at the top of the diagram).

When an upstream set arrives with an incoming message, the timeout value at the head of each queue is compared (in parallel) to the corresponding timeout value in the incoming probe set. If the head's timeout value is less than or equal to the incoming probe set entry's timeout value, the head FIFO entry is removed and its address is sent to the cache to invalidate the stale block. The process is then repeated for the new head of the queue, continuing until either the queue is empty or the timeout value at the head of the queue is larger than the incoming probe set timeout.

In terms of the STAB's physical design, the task of individually decrementing a large set of counters at a constant rate is unattractive in terms of both power and area. It is unattractive in terms of power because the amount of switching is proportional to the number of counter decrements per cycle. It is unattractive in terms of area because it would be difficult to build the STAB out of SRAM if each counter were to be individually decremented (plus a larger area consumes more static power). Consequently, our design uses a slightly different representation that allows for an easy SRAM implementation. Rather than decrementing each counter, only a single counter per FIFO buffer is decremented. This counter (pictured at the bottom of each queue in Figure 6-8), corresponds to the head of the

FIFO. Rather than keep an absolute timer value in each FIFO entry, each entry contains a timeout value relative to the FIFO entry ahead of it. For example, the absolute timer value is shown in parentheses for the rightmost queue from Figure 6-8, and the relative timer values are contained in the queues. In order to calculate this delta value at FIFO insertion, each FIFO additionally maintains a sum of the FIFO's current timeout entries, which is subtracted from the incoming probe timeout value's before insertion. This representation reduces the number of decrements per cycle to the number of other processors in the system. Using a linked list-based queue implementation, we can make efficient use of the dedicated SRAM by allowing the queues to dynamically load-balance, with some queues consuming more than their share when other queues are not filled.

Because the STAB is a finite resource, when its capacity is reached the cache hierarchy can be signaled that an incoming invalidation should trigger a state transition to the I state instead of the ST state. These dropped probes result in subsequent reads to that block being stalled while the data is fetched from across the system, losing performance but still maintaining correctness. In general, the design of the STAB can be quite flexible. In the design presented here, we are able to expire probes in the order in which they arrive, allowing us to continue using stale blocks as long as possible. However, correctness does not depend on the precise relative expiration of supplanter probes with respect to one another. So long as each STAB entry expires no later than the associated probe maintained in the external PPB(s), the protocol will continue to function correctly, allowing for additional implementation flexibility if needed.

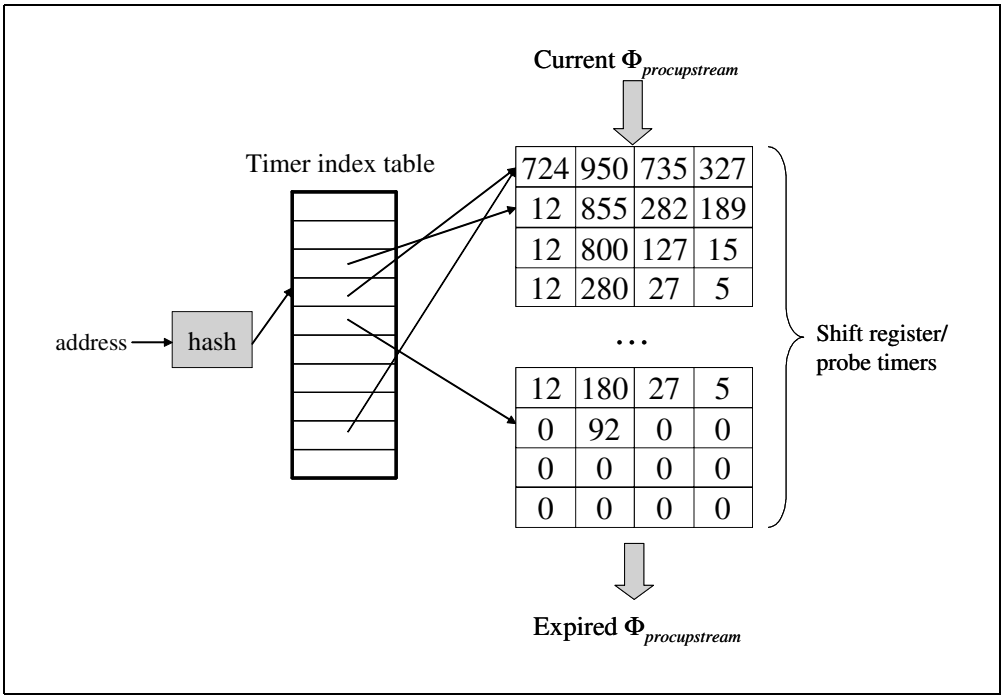

FIGURE 6-9. PPB Organization

6.4.3 Probe propagation buffer (PPB)

Each processor p_i's *probe propagation buffer* (PPB) implements the processor's upstream set $\Phi_{procupstream,i}$ and an approximation of the read upstream set $\Phi_{readupstream,i,a}$ and write upstream set $\Phi_{writeupstream,i,a}$ for each recently touched cache block a. The PPB's purpose is to associate with each address the processor's upstream probe set at the time of the processor's last read or write. Illustrated in Figure 6-9, the PPB consists of two tables: a probe timer table (shown at the right of the figure) and a timer index table. The probe timer table contains snapshots of the processor's upstream probe set at different times in the past and is organized as a FIFO with each FIFO entry containing one probe set (i.e. in an n-processor system, each entry contains n timers). As the processor becomes causally dependent upon new probes (through the arrival of an incoming response message from another processor, or the creation of a new probe due to a local store miss), the

new probes are added to the tail of the FIFO (at the top of the figure). As probe sets expire, they are removed from the head of the FIFO (at the bottom of the figure). Consequently, the tail of the probe timer table always contains the processor's current upstream set.

The timer index table implements a mapping from cache block physical address to the entry in the probe timer table containing the processor's probe set at the last time that address was read or written. Each load or store instruction accesses the timer index table (through the hash function), storing a pointer to the tail of the probe timer table at the hashed entry. When response messages are sent from this processor to other processors, a timer index table lookup occurs using the physical block address of the coherence message, and the corresponding entry from the probe timer table is read and attached to the outgoing message. Because these coherence message induced lookups may need to read any entry (not just the head or the tail), each timer in each entry must maintain a precise timer value (decremented every cycle) to allow easy attachment to the outgoing message.

Because there is no mapping from a probe timer table entry back to the probe index table entries which point to it, when a probe expires, it is not possible to reset those entries in the timer index table that point to the timer table entry. Consequently, those index table entries will continue to point to the deallocated entry, even after is has been reallocated for a new probe set. This could create a performance problem, because once the timer index table is full of mappings, if the probe timer table is full, it will appear that every address has a mapping to a live probe set. We probabilistically solve this problem by augmenting timer index table

entries with a tag that identifies the current wrap-around count for the probe timer table. The probe timer table maintains this count using a single non-saturating counter that is incremented each time the timer table FIFO wraps around. Timer index table entries consist of the concatenation of this timer and the index of the probe timer table. When performing a PPB lookup, this portion of the index table entry is compared to the current value of the timer table wrap-around counter. Based on the current position of the PPB head pointer and this comparison, it can be determined if the PPB index table entry truly points to the contents of the corresponding PPB timer table entry.

In an ideal PPB implementation (unconstrained by physical resources), the timer index table would be large enough to store two mappings from address to upstream set for every address touched until the oldest probe inherited by the processor expires. One mapping would map the address to the probe set at the last time the block was touched by a read, the other for the last time it was touched by a write (thus implementing the $\Phi_{readupstream,i,a}$ and $\Phi_{writeupstream,i,a}$ for a block a). Due to physical size limitations, an approximation of the $\Phi_{readupstream,i,a}$ and $\Phi_{writeupstream,i,a}$ sets is maintained, by using a single mapping for each block address (coalescing the location's $\Phi_{readupstream,i,a}$ and $\Phi_{writeupstream,i,a}$ sets). In Chapter 7 we evaluate the effect of this decision, and find that there is little to be gained by maintaining separate mappings.

Because there are fewer timer index table entries than addresses touched during the lifetime of a probe set (and the hash function is imperfect), there will be collisions in the table, causing some addresses to be mistakenly associated with

"newer" probe sets than the addresses should be associated. Because all such errors occur in the correct direction (i.e. addresses are never mistakenly associated with probe sets older than they should be), this physical limitation may affect performance but not correctness.

The probe timer table is also limited in size, limiting the number of unique probe sets that can be associated with block addresses. Consequently, as probes are added to a processor's upstream set, a new probe timer table entry cannot always be allocated. When the timer table is full, we continue adding new probes to the current tail entry, by performing a *max* operation on the current tail entry and incoming probe set. Once the probes in the probe set at the head of the probe timer table expire, then subsequently arriving probes will be added to a new tail entry. This policy causes cache blocks that were previously associated with that probe set to become causally dependent upon the newly added probes, when they do not need to be, but enables a compact and correct implementation.

Another, smaller, PPB called the CastoutPPB is maintained at the directory controller. Because we are using a directory protocol in which dirty data is written back to memory when servicing a read request, writebacks occur more frequently than they do in most snoop-based systems. Rather than assume that any read serviced from memory returns data that causally precedes all of the supplanting write probes in the STAB (wiping out all of the stale blocks), we add a smaller version of the PPB at the directory controller. The version at the directory controller is smaller than the processor-side PPB because such writebacks still occur much less frequently than the loads and stores that update the processor-side PPB.

This method of tracking causality is similar to the use of vector clocks associated with cache blocks in the lazy-release consistency protocol, and in recent work detecting violations of sequential consistency [21][51]. Xu, Bodik, and Hill use a similar method based on a scalar clock in which a last-touch instruction count is associated with cache blocks and passed among processors in order to track inter-processor dependences [112].

6.4.4 Other aspects of the ECDC implementation

In this section, we describe a few additional details that were not discussed in the previous sections.

6.4.4.1 Critical write detection

Because delaying the observance of writes until a stale block's probe timer expires would sometimes decrease performance by delaying time-critical writes such as lock releases, we use two heuristic mechanisms to detect such situations. In the absence of accurate detection, the eventual expiration of the probe timer ensures that the application will continue to make forward progress.

The first heuristic detects potential critical cache blocks by assuming that the observance of a write to any block that has been previously touched by a store conditional instruction should not be delayed. To implement this heuristic, we augment each cache block with a single bit, set when the block is touched by a store conditional. This bit is communicated with the block data when the cache block is transferred among processors. The bit is not stored in memory when the block is cast out, allowing the bit to be reset periodically (so that a block will not always be associated with synchronization after it is reallocated for another purpose). When a

stale cache block is accessed whose bit is set, the processor can continue to use the stale block, but a coherence transaction is immediately initiated to fetch the new copy of the block.

The second heuristic is used to detect polling behavior to cache blocks that have never been touched by a store conditional instruction. Should two read requests to the same physical address from the same static instruction be issued to the data cache consecutively, we conservatively assume that the processor is polling that location for a new value. The cache may return a stale value to the request, but immediately issues a coherence transaction to obtain the new copy of the block.

For the applications used in this thesis, we find that these heuristics perform quite well, because the applications do not suffer a performance penalty (or else the performance penalty is offset by performance gains from using stale data), as will be presented in the next chapter.

6.4.4.2 Atomic synchronization primitives

The ECDC protocol has little effect on the implementation of atomic synchronization primitives. Primitives such as test & set, and fetch & op are handled identically to a typical coherence protocol, in which an exclusive copy of the block must be obtained prior to completing the operation. Should an implementation of compare and swap delay obtaining an exclusive copy of a block until after a successful comparison, we recommend that this comparison not be based on stale data for reasons of forward progress. When implementing PowerPC's load-linked synchronization primitive in our infrastructure, we allow the load linked to return stale

data but we do not set the lock register. A coherence transaction is immediately issued for the new copy of the block.

6.4.4.3 Interaction with speculative loads

Should an implementation detect memory ordering violations using a snooping load queue, the load queue must be snooped when a block transitions from the stale state to invalid just as when it transitions to invalid from other states. This policy prevents a case in which a speculative load instruction may read stale data, followed by an earlier instruction becoming causally dependent upon a probe that forces the stale block to transition to the I state, in which case the load instruction should not have read the stale data. When a block initially transitions to the ST state, there is no need to snoop the load queue, because the ST state is essentially the same as the S state.

6.4.5 Examples of operation

In this section we describe the protocol events and actions that are affected by the ECDC implementation. Supporting ECDC requires no changes to the original directory protocol other than 1) the addition of *ProbePropagate* and *ProbePropagateResponse* messages, which require no ordering properties with respect to any other coherence messages, 2) the addition of the ST state to each cache hierarchy, which functions identically to the shared state (except initiating a miss when touching a potential synchronization block), and 3) the augmentation of protocol response messages (other than NACKs) with probe sets.

Our discussion here focuses on two common scenarios: 1) a store miss to a cache block in shared state at other processors; and 2) a load miss to a block that is

Table 6-3: Actions taken in typical handling of store miss

Event	Action(s)
1. Store miss	1. Requestor sends ReadExclusive message to home node. **2. Requestor adds new probe to its PPB corresponding to the block address.**
2(a). Reception of ReadExclusive message by directory controller at home node, with block in the shared state	1. Send invalidation message to each sharer. 2. Send response to requestor, with number of sharers and data. 3. Reset list of sharers, set block to owned state with requestor as owner.
2(b), (c) Reception of Invalidate message by sharer	**1. If present, transition block to the ST state.** (Rather than the I state in typical protocol.) **2. If present, add entry to the STAB for the incoming invalidation using the stale block lifetime timeout value minus the maximum transit time.** 3. Send invalidate acknowledgment message to requestor, **augmented with the block's upstream set obtained from the PPB.**
3(a), 3(b) Reception of invalidate acknowledgment messages from sharers	**1. Lookup STAB using incoming probe set, transition blocks from ST to I state as needed.** **2. Update PPB's timer index table with block address.** 3. Transition block to modified state, notify processor of store completion.

dirty in another processor's cache.

Figure 6-10 illustrates the messages passed in a typical directory-based coherence protocol as a result of a store miss to a cache block in the shared state at the directory. In this example the block is shared by two other processors S_1 and S_2. The ECDC protocol uses the same set of messages, but augments the messages with probe sets depending on the current contents of the PPB. Table 6-4 describes the messages and event handlers for each transition, with modifications made for the ECDC protocol listed in bold.

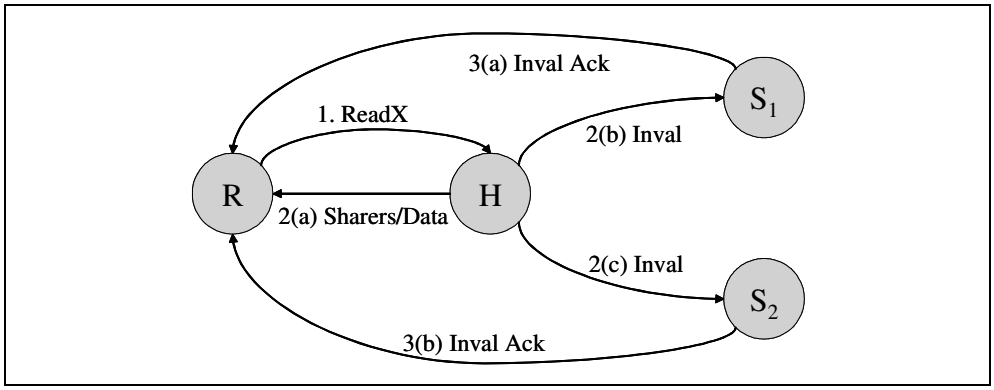

FIGURE 6-10. Read Exclusive Request Events ((R)equestor, (H)ome, (S)harer$_1$, (S)harer$_2$)

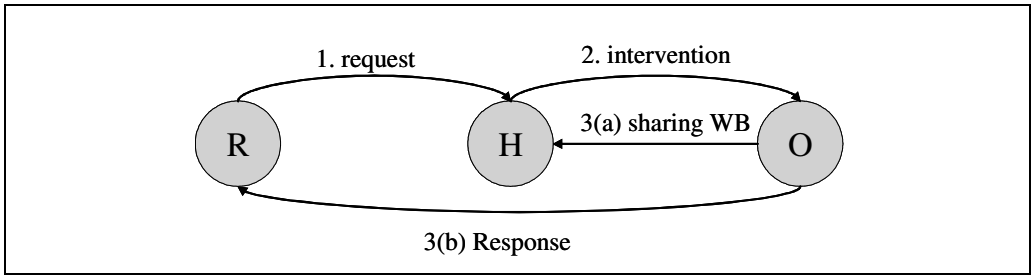

FIGURE 6-11. ReadShared to Block in Modified State ((R)equestor, (H)ome, (O)wner)

All other write miss events are handled similarly. When receiving an upgrade transaction, the directory responds with the number of sharers and probe set, but no data. If the block is in the modified state at the directory, the request is forwarded to the owner, who similarly responds with data and an upstream set. If the block is in the invalid state at the directory, the data is returned with an upstream set, should one exist in the CastoutPPB.

A typical load miss that finds the directory in the modified state, illustrated in Figure 6-11, is handled similarly to the occurrence of write misses that reach the directory in the modified state. Table 6-4 specifies the set of events and actions, with changes needed for supporting ECDC in bold. In this case, when the owner of the block sends a sharing writeback message to the directory, it attaches the block's upstream set to the message, which is subsequently saved in the Castout-

Table 6-4: Actions taken in typical dirty miss handling

Event	Action(s)
1. Load miss	1. Send ReadShared request to home node
2. Reception of ReadShared request by directory controller at home node	1. Forward ReadShared to owner on intervention network 2. Add requestor to list of sharers 3. Set block state to BusyShared
3(a). Reception of ReadShared request by owner	1. Set the block's state to shared 2. Send response with data to the requestor, **augmented with the block's upstream set obtained from the PPB.** 3. Send SharingWB message to home node**, augmented with the block's upstream set obtained from the PPB.**
3(a). Reception of SharingWB by directory controller at home node	**1. Add upstream set attached to incoming message to the CastoutPPB entry for that block** 2. Write data back to memory. 3. Transition the block to the shared state.
3(b). Reception of response message by requestor	**1. Add upstream set attached to incoming message to the tail PPB entry.** 2. Install cache block in the shared state.

PPB when received by the directory.

This concludes our ECDC implementation description. In the next section, we discuss prior work related to delayed consistency and the edge-chasing delayed consistency protocol that was not discussed earlier. In Chapter 7, we evaluate the proposed implementation.

6.5 Related Work

There has been a significant amount of related work on mechanisms that prevent the performance penalty associated with inter-thread communication, including optimizations at the algorithm level, language level, compiler level, and run-time system/hardware implementation level. The discussion here will be lim-

ited to those techniques that affect the shared-memory implementation, whether it be a hardware-based implementation or software-based implementation. We limit this discussion to other single-writer protocols that improve communication performance through the use of stale values or by delaying the observance of writes, and related work that specifically targets the false sharing problem.

6.5.1 Hardware systems

Edge-chasing delayed consistency is closely related to prior implementations of delayed consistency. Motivated by the problem of false sharing, Dubois et al. proposed the first delayed consistency protocols, which delayed either the sending of all invalidates (sender-delayed protocols) or the application of all invalidates (receiver-delayed protocols) or both until a processor performs a synchronization operation [34][36]. Their work found significant reductions in cache miss rates from delayed consistency, however their studies did not determine if performance benefits could be obtained from these reductions. Dahlgren and Stenstrom more thoroughly explore sender delayed protocols implemented through the addition of a write cache that buffers outbound invalidate messages until an acquire or release is performed [32]. Their work focuses on update protocols and hybrid update/invalidate protocols. These proposals have demonstrated a reduction in multiprocessor coherence misses, but unfortunately each relies on properly-labeled synchronization operations[1]. Edge-chasing delayed consistency

1. With the exception of IA-64, there are no commercial instruction set architectures which require using special labeled operations for synchronization. In the majority of commercial architectures, it is possible to implement synchronization using ordinary load and store operations. Consequently, it is not advisable for an implementation to delay such operations, limiting the practicality of previous delayed consistency mechanisms.

overcomes this obstacle, through heuristics that capture common synchronization constructs, and timeout mechanisms to prevent permanently delaying write observance for synchronization that is not captured by the heuristics. In addition, edge chasing delayed consistency extends the useful lifetime of stale cache blocks by allowing a processor to use them until that processor becomes causally dependent on a newer copy of the cache block. In this sense, edge-chasing delayed consistency approximates the behavior of the entry consistency model, which orders operations that are related to one another (e.g., stores to the same data structure), while eliminating ordering requirements for operations that don't need to be ordered [12]. In comparison, the work by Dubois et al. and Dahlgren and Stenstrom take an all or nothing approach to the delaying of writes, in which all writes are delayed until a synchronization operation occurs, after which all pending invalidations are applied. The prior studies by Dubois et al. and Dahlgren and Stenstrom have also been limited in terms of experimental methodology. Demonstrating a reduction in miss rates is a positive outcome, however such reductions do not necessarily lead to performance improvement.

Lebeck and Wood's work on dynamic self-invalidation also included support for a form of receiver-delayed consistency by marking certain blocks in a directory-based coherence protocol as "tear-off blocks", which would be automatically invalidated by the cache controller upon the next miss (under sequential consistency) or the next synchronizing operation (under weak ordering) [61]. Until a miss or synchronization, a processor could continue to use the potentially stale copy of the block. The tear-off blocks would not need to be tracked by the direc-

tory controller, avoiding the transmission of invalidation messages when the block was written by another processor. This work focused on the reduction in write latency gained by shortening the list of sharers and thus avoiding the corresponding invalidation/acknowledgment latency; the extent of benefit obtained from lengthening the useful lifetime of cache blocks is unclear.

Lepak evaluates a mechanism that speculatively returns stale data on a coherence miss, allowing the processor to continue executing dependent instructions until the cache miss returns and the speculation has been verified [63]. Huh et al. also describe a class of speculative protocols which include the mechanism proposed by Lepak [48]. Their mechanism increases the accuracy of stale data speculation by occasionally updating the stale data or by using a confidence predictor to decrease the number of misspeculations. These speculative protocols are complementary to the edge-chasing delayed consistency mechanism presented here if used by a processor that supports speculation. When the edge-chasing consistency mechanism indicates that a stale cache block can be used, it can be used non-speculatively, allowing the processor to commit the instruction, whereas the speculative mechanism would have forced the instruction to wait for the cache miss to return, potentially stalling the machine. The speculative mechanism will be useful in cases where the edge-chasing consistency mechanism indicates that it is not safe to use stale data. In the context of in-order processors that do not support a method for speculative run-ahead, only non-speculative schemes like the edge-chasing delayed consistency mechanism can provide performance benefit.

Rajwar, Kagi, and Goodman describe a delayed mechanism of a different

type that enables the use of exclusive prefetches for load-linked operations by delaying the transfer of ownership until the subsequent store-conditional operation completes, thereby improving the throughput of synchronization [92]. Like edge-chasing delayed consistency, this optimization removes the overheads of some communication by delaying the visibility of writes. However, because it is tailored specifically to synchronization variables it can exploit common synchronization variable access patterns (acquire->release->acquire...), but its usefulness is also limited to synchronization variables.

There have been several proposed hardware mechanisms that attack the false sharing problem in addition to the work described above. Dubnicki and Leblanc evaluate a coherence protocol that utilizes an adjustable block size, dynamically detecting false sharing misses and splitting cache blocks in half accordingly [33]. Chen and Dubois propose a sub-blocked cache in which coherence messages invalidate only part of a cache block, allowing other parts of the cache block to be used [25]. Anderson and Baer propose a similar protocol that uses a large transfer size and small coherence unit in order to take advantage of spatial locality while avoiding false sharing [9]. Each of these proposals is successful at reducing false sharing misses, however their benefit is limited by the amount of false sharing in the application. Edge-chasing delayed consistency can eliminate false sharing misses, but it can also reduce misses to truly shared data as well.

Afek et al. and Brown describe theoretical delayed consistency algorithms similar to Dubois et al.'s in the context of update-based coherence protocols and invalidate-based protocols, respectively [2][16]. These algorithms limit a proces-

sor's use of stale values to those that are more recent than the most recent write by that processor. Their work includes formal presentations of the caching algorithms, but does not include any experimental evaluation.

Rechtschaffen and Ekanadham describe a cache coherence protocol that allows a processor to modify a cache line while other processors continue to use a stale copy [93]. Their protocol ensures correctness by conservatively constraining certain processors from reading stale data or performing a write while stale copies exist. Published only in patent form, their proposal is accompanied by no experimental data. Their proposal is limited by the effectiveness of heuristics that dictate when stale data can be used. Because edge-chasing delayed consistency tracks the constraint graph, it can better identify when stale data can or cannot be used, and it should be able to more frequently use stale data than the more approximate method proposed by Rechtschaffen and Ekanadham.

6.5.2 Software systems

Keleher, Cox, and Zwaenepoel's lazy release consistency protocol is a sender delayed protocol that is similar to the receiver-delayed edge-chasing implementation discussed here [51]. Like edge-chasing delayed consistency, Keleher's proposal delays the observance of writes until a processor's read operation becomes causally dependent upon a write, at which point all writes that causally precede the observed write will become observable to the reading processor. Like the hardware-based delayed consistency work, lazy release consistency is dependent upon properly labeled synchronization operations, limiting its applicability to current architectures. Also, because lazy release consistency was proposed and

evaluated in the context of a software-based distributed memory machine managing coherence granularity at the size of a page, the trade-offs involved are quite different from those involved in a hardware based multiprocessor. An interesting avenue of future work would be to compare the overheads of a software-based implementation of edge-chasing delayed consistency to lazy release consistency.

Tambat and Vajapayem present the performance advantages of a non-blocking memory access primitive called *Global_Read* in the context of a software-based distributed shared memory machine [106]. The Global_Read primitive returns new data once communication has completed, but until that time returns the previous copy of the data, allowing the application make forward progress using stale data until the new data is locally available. The mechanism is targeted at data-race tolerant applications such as iterative equation solvers, genetic algorithms, and probabilistic inference in Bayesian belief networks. At a high-level, this mechanism is similar to the edge-chasing delayed consistency mechanism presented here, because they both allow stale shared data to be read in order to tolerate communication latency. However, the Global_Read primitive provides an explicit interface to the programmer who can then dictate whether or not stale data should be used for a particular access. This is a powerful mechanism, but places a significant burden on the programmer, whereas the ECDC mechanism potentially improves the performance of arbitrary programs without programmer intervention.

Wang and Weihl describe a software caching algorithm used in an implementation of concurrent B-trees that allows multiple versions of memory such that local stale versions of the B-tree can be read without incurring a cache miss (simi-

lar to the example in Section 6.1.1), saving the latency of fetching the new version [109]. Their implementation improves performance over 300% for a highly contended B-tree microbenchmark. The ECDC protocol can provide similar performance benefits as multi-version memory for a similar microbenchmark (which we show in Chapter 7), however ECDC is a more general solution, creating benefits for unmodified programs when potential exists, and requiring no additional programmer effort.

Chapter 7

Experimental Evaluation of Edge-Chasing Delayed Consistency

In this chapter, we present a detailed performance evaluation of the edge-chasing delayed consistency mechanism. We compare the protocol to a baseline machine utilizing a conventional directory coherence protocol, based on the SGI Origin [60], whose configuration is detailed in In Section 7.1. In Section 7.2, we characterize the relevant behavior of coherence misses across a set of scientific and commercial applications to gauge the opportunity for performance gains from edge-chasing delayed consistency and to provide insight into the subsequent evaluation. In Section 7.3, we evaluate the ECDC protocol assuming unlimited probe timeouts, with unlimited STAB and PPB storage space, to determine its potential without being subject to physical constraints. We complete our evaluation in Section 7.4 with a study of the protocol's performance when using realistic resource constraints, followed by a summary in Section 7.5.

7.1 Machine Configuration

Table 7-1 describes the machine configuration used for these experiments. We use a four-processor baseline machine, whose processor core and cache hierarchy (except for the larger L3 cache) are identical to the baseline configuration used Chapter 5's value-based memory ordering evaluation. The modeled interconnect topology (and latencies and bandwidths) are based on the Alpha 21364 network [84]. We replace the 21364's dynamic routing protocol with a simpler static

Table 7-1: Baseline machine configuration.

Out-of-order execution	5.0 GHZ, 15-stage 8-wide pipeline, 256 entry reorder buffer, 128 entry load/store queue, 32 entry issue queue, store-set predictor with 4k entry SSIT and 128 entry LFST.
Functional Units (latency)	8 integer ALUs (1), 3 integer MULT/DIV (3/12), 4 floating point ALUs (4), 4 floating point MULT/DIV (4, 4), 4 L1D ports
Front-end	fetch stops at first taken branch in cycle, combined bimodal (16k entry)/gshare (16k entry) with selector (16k entry) branch prediction, 64 entry RAS, 8k entry 4-way BTB
Cache hierarchy (latency)	32k direct-mapped IL1 (1), 32k direct-mapped DL1 (1), 64 entry write buffer, 512k 8-way DL2 (7), 512k 8-way IL2 (7), Unified 16MB 8-way L3 (15), 128 byte cache lines. 2k entry 2-way ITLB, 2k entry 2-way DTLB. Stride-based prefetcher modeled after Power4.
Interconnect/Memory	2-D torus static dimension order routed interconnect. 15 ns (60 cycle) per link+route (40GB/S bandwidth) 400 cycles/100 ns best-case DRAM latency. 10 cycle directory access latency

dimension-ordered routing mechanism.

7.2 Coherence Miss Characterization

Figure 7-1 shows the number of misses per 1000 committed instructions for a set of parallel applications (this graph is repeated from Chapter 6). Each bar is broken into its cold, coherence, and capacity/conflict components [47]. The top of each bar additionally includes upgrade transactions, caused by writes that touch a shared copy of the block, creating inter-processor communication but no data transfer. Many of the applications incur a significant number of coherence misses, especially the four commercial workloads at the right side of the figure. Such misses cause significant performance penalties, particularly in home-based protocols where they must typically make three network hops: from the requester to the home node, from the home node to the current owner, and back to the requester.

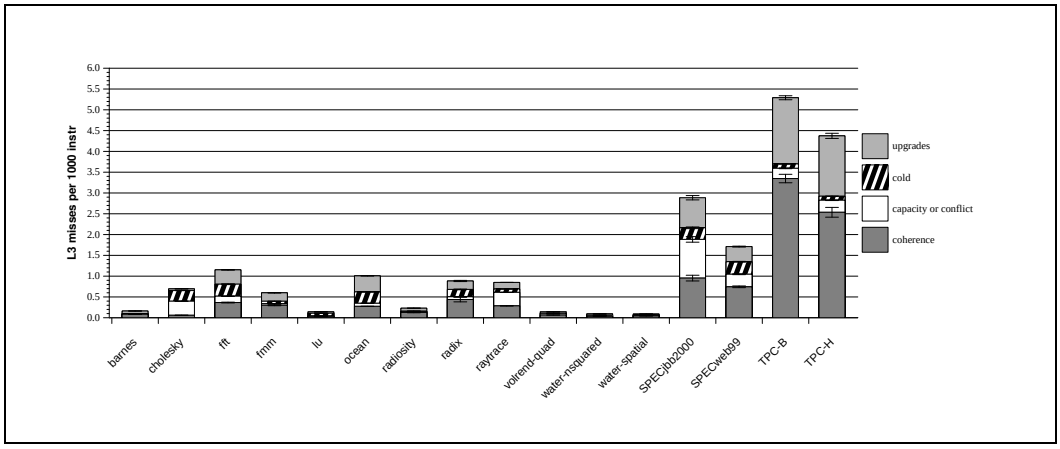

FIGURE 7-1. Misses per 1000 committed instructions for 16MB L3 cache.

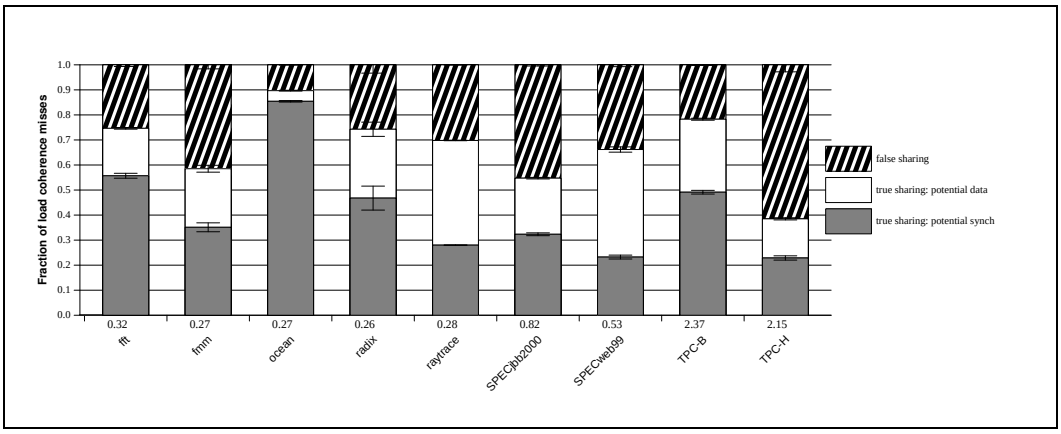

FIGURE 7-2. Breakdown of coherence misses caused by load instructions. (The number of load coherence misses per 1000 committed instructions is labeled beneath each bar).

Because some of the applications do not incur many coherence misses (barnes, cholesky, lu, radiosity, volrend, water-nsquared, and water-spatial), we omit these applications from the rest of our results. We do not expect edge-chasing delayed consistency to significantly improve their performance.

Edge-chasing delayed consistency should benefit applications most by reducing the average latency of load instructions, because write buffers are able to hide most of the performance degradation caused by upgrade misses for these applications. Figure 7-2 further breaks down those coherence misses caused by load instructions into three categories: false sharing misses, true sharing misses

that reference potential synchronization memory locations, and true sharing misses that reference potential false sharing memory locations. False sharing and true sharing misses are differentiated using the Dubois classification [35]. We separate true sharing misses into potential data misses and potential synchronization misses by labeling a miss as potential synchronization if the referenced cache block has been touched by a load-linked or store conditional instruction during the simulation, and all other misses are labeled as data. This classification is only approximate, because a memory location that is used once for synchronization may later be reallocated for a different purpose, but will still be considered synchronization using this classification. Consequently, the number of misses labeled synchronization may be overestimated, but we expect the classification to be useful as a rough estimate.

We expect edge-chasing delayed consistency to offer performance improvement for those misses that are caused by false-sharing, and for some truly shared misses to data. ECDC should not offer any performance improvement by reducing misses to truly shared synchronization data (the black portion of each bar), because these misses are likely fetching the release of a lock variable. Although this class of misses is significant for each application, it represents no more than half of all load coherence misses for any applications other than fft and ocean. At 73%, the TPC-H decision support benchmark contains the largest percentage of misses caused by false sharing and true data sharing.

This data indicates that coherence misses occur frequently enough that their avoidance should yield some performance benefits, particularly in the com-

mercial applications SPECjbb2000, SPECweb99, TPC-H and TPC-B. Because ECDC must maintain extra state for cache blocks while they are in stale state, it is also important to determine the amount of time that typically passes between the arrival of an invalidation for a cache block and a subsequent reference to that cache block. Figure 7-3, Figure 7-4, and Figure 7-5 chart the cumulative distribution of this distance (in cycles) for all blocks, potential synchronization blocks, and potential data blocks, respectively, that are subsequently referenced by the processor. In Figure 7-4, for example, in raytrace 88% of the invalidated cache blocks that are subsequently referenced are referenced within 10,000 cycles of their invalidation (note log scale on x-axis).

In general, this distance can be quite large. When looking at the distribution for all cache blocks in Figure 7-3, it requires 100 million cycles to capture 90% of all of the coherence misses across all applications. There is significant variability between applications, however; for instance raytrace requires only a 100,000 cycle window to capture 90% of the load coherence misses. When breaking this chart into potential synchronization and potential data components in Figure 7-4, and Figure 7-5, we find that the behavior is quite different between the two. The curves for potential synchronization cache blocks generally rise more quickly than the curves for potential data. Presumably, this is a result of contended lock locations which are likely to be read soon after being written. TPC-H is an exception to this rule, demonstrating the complete opposite behavior, as the curve for potential data cache blocks rises significantly earlier than the curve for potential synchronization cache blocks. In fact, for TPC-H, over 90% of potential data

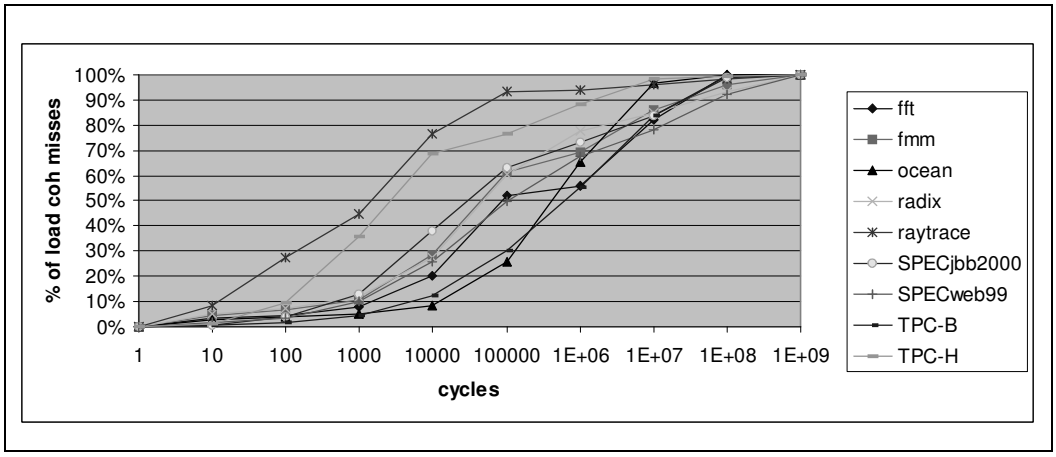

FIGURE 7-3. Invalidation to miss cumulative distribution (All load coherence misses).

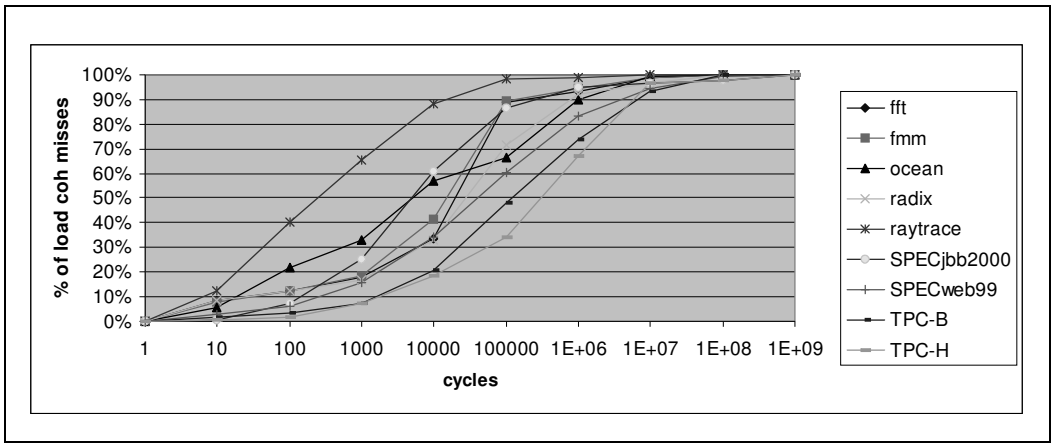

FIGURE 7-4. Invalidation to miss cumulative distribution (Potential synchronization misses).

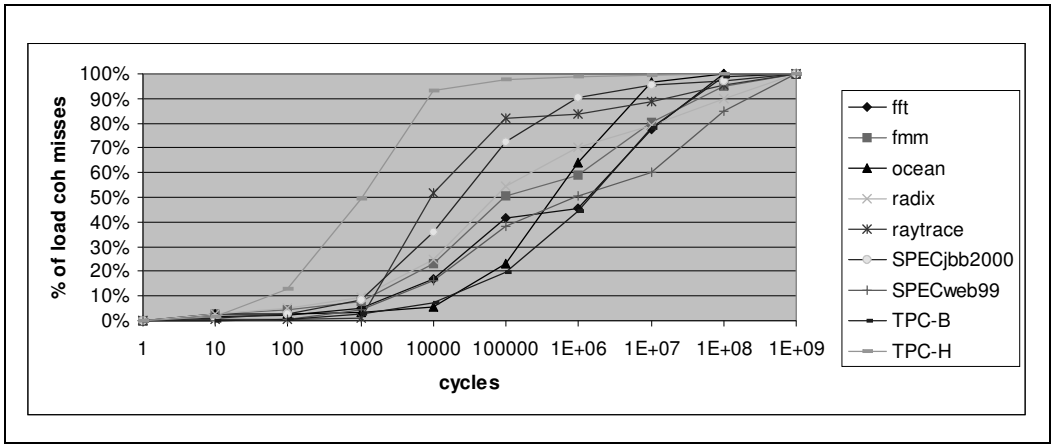

FIGURE 7-5. Invalidation to miss cumulative distribution (Potential data misses).

load coherence misses are initiated within 10,000 cycles of the block's invalidation. ECDC should perform best for those applications whose potential data coher-

ence misses occur soon after the block's invalidation; consequently, it should work well for TPC-H.

7.3 ECDC Performance: Unlimited Stale Block Lifetime

In the previous section, we demonstrated that there is potential for the ECDC protocol should it be able to keep a block in the stale state long enough to capture its next reference. In this section, we evaluate a few variations of the ECDC protocol when assuming an infinite STAB and PPB, and an unlimited time-out for each probe, to gauge the potential for this technique without being subject to resource constraints. Later in the chapter, we evaluate ECDC while placing realistic assumptions on these parameters. Our evaluation is broken into two parts, a microbenchmark evaluation and an evaluation using the applications characterized above.

7.3.1 Microbenchmark evaluation

As described in Chapter 6, edge-chasing delayed consistency offers performance improvement potential for parallel applications that share linked data structures. In this section, we compare the performance of ECDC to a conventional coherence protocol when running a lock-free list insertion microbenchmark, in the context of a 16-processor machine. We use Michael's hazard pointer-based lock-free parallel list maintenance algorithm for our microbenchmark's implementation [78].

The microbenchmark consists of a set of threads randomly inserting, deleting, or searching a linked list with some probability, where the probability x of a

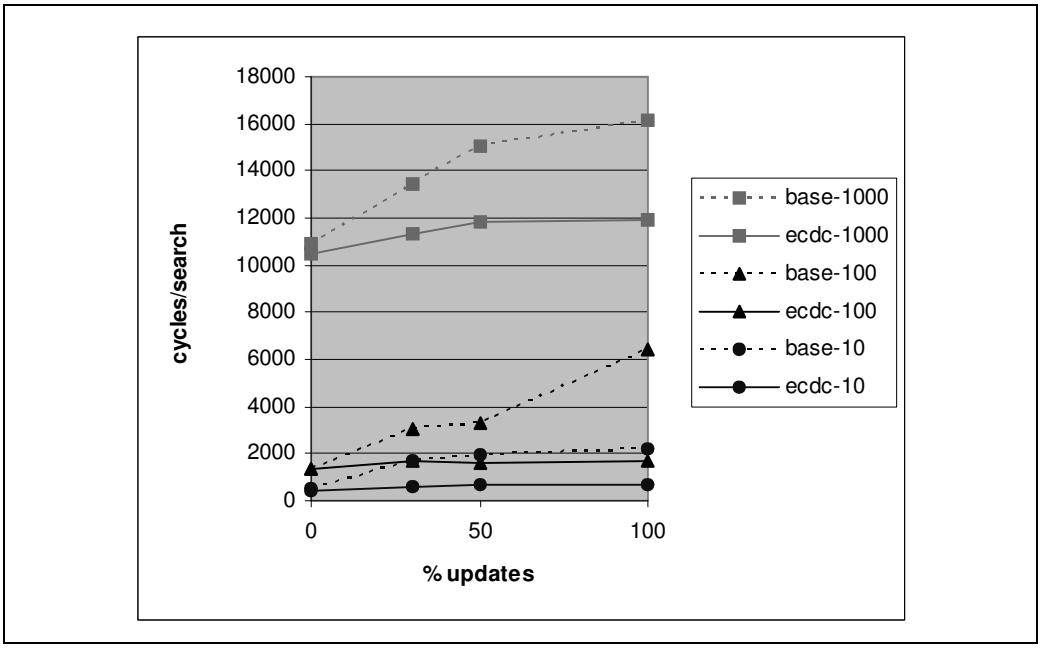

FIGURE 7-6. Lock-free list insertion microbenchmark performance.

list insertion is always the same as the probability of a deletion. Each operation randomly chooses a node for which it will search, delete, or insert a new subsequent node. We use 15 threads running this mix of operations, and a single thread whose search operation latency is timed. Figure 7-6 charts the average list search latency, varying the *x* parameter from 0 to 50, resulting in the percentage of list modification operations ranging from 0% to 100%. The test was performed using three different average list lengths: 10, 100, and 1000, with larger list lengths decreasing the amount of contention in the microbenchmark.

As one would expect, as the fraction of update operations increases the average search time for the baseline machine also increases. For the highly contended list of length 10, the time per search increases by a factor of 4.2. As the fraction of update operations increases, the performance levels off; a point is reached with such a short list length at which contention is high enough that the probability of a cache miss occurring no longer increases. This is not true for the

longer list lengths, where performance continues to degrade as the fraction of updates increases. For the list length of 100, the performance with 100% updates is 4.7 times worse than the performance with no updates. Moving to the 1000 entry list, performance is less affected by the updates because there is less contention, but degradation is still significant (48% worse performance) when moving from no updates to 100% updates.

When using the ECDC protocol, performance stays relatively flat as the percentage of updates is increased, because list searches are able to avoid many coherence misses while traversing the list. The performance is not completely flat, because some misses inevitably occur, creating a causal dependence on a recent write that forces many of the reader's stale blocks to the invalid state. However, the ECDC protocol obtains significant speedups relative to the conventional protocol, measuring 2.74, 1.82, and 1.18 for the list of length 10, 100, and 1000 respectively, when 30% of the operations are updates. When 100% of the operations are updates, the ECDC protocol improves performance even more, with speedups of 3.11, 3.87, and 1.35 for these list lengths.

7.3.2 Application evaluation

In this section, we evaluate the performance of three variations of the ECDC protocol relative to a conventional coherence protocol. In addition to the full-blown ECDC protocol (labeled *ECDC base* in each chart), we also evaluate a variation in which we maintain a single probe set per memory location by using a single timer index table mapping in the PPB (labeled *ECDC merged rw sets*), rather than the two mappings that the base ECDC protocol uses to precisely imple-

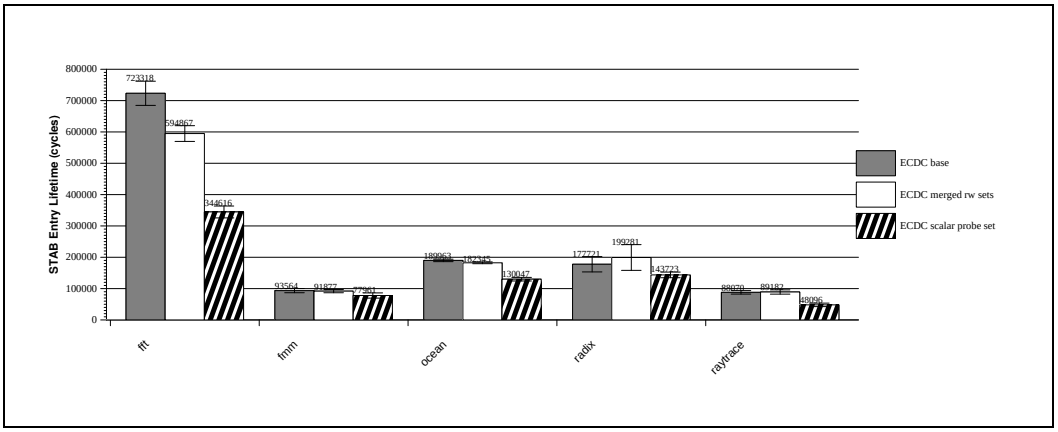

FIGURE 7-7. Average STAB entry lifetime for scientific applications.

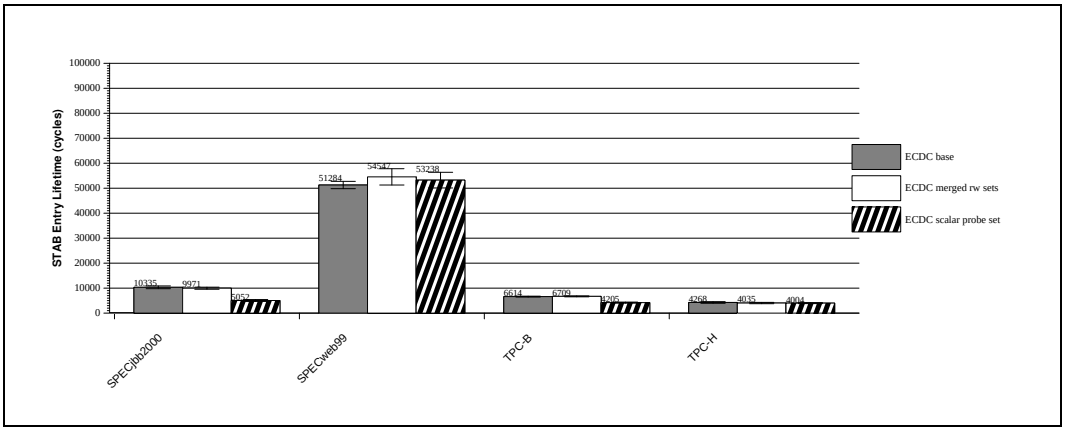

FIGURE 7-8. Average STAB entry lifetime for commercial applications. (Note different y-axis scale.)

ment the read and write upstream sets for each block. We also evaluate a variation of the protocol that uses a scalar timeout value to represent probe sets (labeled *ECDC scalar probe set*), rather than maintaining a vector of entries to individually track a processor's causal dependences on every other processor.

Figure 7-7 and Figure 7-8 illustrate the average lifetime of a STAB entry from its allocation to its deallocation for scientific applications and commercial applications, respectively. (The mean STAB entry lifetime is labeled above each bar.) Because this machine configuration uses an infinite timer, entries never expire due to a timeout, they may only expire due to the processor becoming causally dependent upon the supplanting write to which the STAB entry corresponds.

The average STAB entry lifetime varies greatly among applications, ranging from a maximum average of more than 700,000 cycles to a minimum of 4000 cycles. As one would expect, as the ECDC implementation loses its granularity by first moving from two probe set mappings to a single mapping, and then from the vector probe set to the scalar probe set, the average lifetime of each STAB entry decreases because the implementation loses its precision, rounding conservatively. This is the case for all applications, although for most applications the difference between the base ECDC configuration and the single mapping configuration is small enough that it is within the margin of error caused by application non-determinism.

A measure of the protocol's ability to use stale data is presented in Figure 7-9. We define an intolerable load miss as those load misses to blocks in the invalid state. A reference to a stale block is tolerable because it returns stale data, but the reference may also initiate a coherence transaction (if it is determined to be a synchronization reference). Only part of this reduction in intolerable misses will result in improved performance, because for some of these misses the processor may simply continue to poll a memory location waiting for a new value to appear, without accomplishing any useful work. Each bar in Figure 7-9 is broken into true sharing misses and false sharing misses, and the true sharing component is further broken into potential synchronization and potential data misses. The rightmost three bars correspond to the bars in the prior figure (ECDC base, ECDC merged r/w sets, ECDC scalar), and the left-most bar corresponds to the baseline conventional machine.

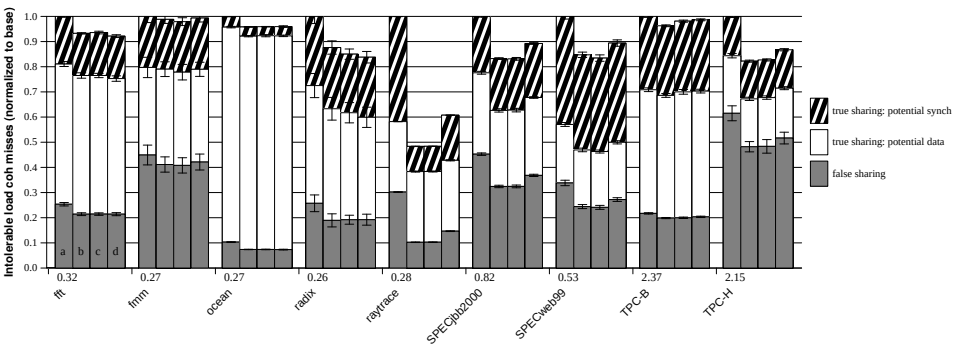

FIGURE 7-9. Reduction in intolerable load coherence misses. (a) baseline (b) ECDC base (c) ECDC with merged r/w sets (d) ECDC with scalar probe sets. The number of load coherence misses per 1000 instructions for baseline is labeled beneath each bar.

We find that for many applications a significant fraction of intolerable misses is removed when using the ECDC protocol. In raytrace, the application with the largest reduction, as many as 52% of the intolerable misses are eliminated for the base ECDC protocol. However, approximately half of these misses are potential synchronization to truly shared data, which will probably not yield performance improvement. The other half of the reduction comes from false sharing misses. Across the remainder of the applications, nearly all of the reduction comes from these categories; there is very little reduction in misses to truly shared data. It is our understanding that AIX does not use any lock-free algorithms, and this set of applications does not include any convergent iterative algorithms, in which we would expect to observe a reduction in intolerable misses to truly shared data. From this data, it appears that any performance gains from the ECDC protocol will come from its reductions in false sharing miss, for which there is a significant amount for many of the applications.

In terms of the relative ability of each of the three ECDC implementations to avoid misses, the ECDC protocol with merged read and write sets performs sim-

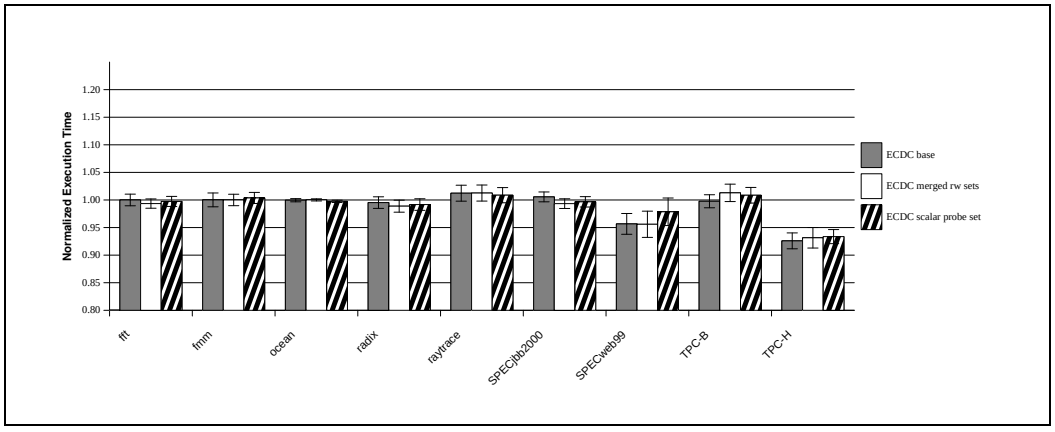

FIGURE 7-10. ECDC performance relative to baseline.

ilarly to the base ECDC protocol across all of the applications. The scalar ECDC implementation, although sacrificing some of the gains from the base ECDC implementation, still performs quite well despite the significant reduction in state from the vector-based representation.

Figure 7-10 presents the most important metric, the normalized execution time of each of the three ECDC variations relative to the baseline machine. Unfortunately, for most applications, the ECDC protocol has little effect on performance. There are two applications, SPECweb99 and TPC-H, in which the ECDC protocol offers measurable improvements in performance (4% and 8%, respectively, for the base ECDC implementation). Of all the applications, ECDC should improve TPC-H most, because TPC-H has a significant number of coherence misses, most of which are caused by false sharing, and of all the applications its coherence misses occur the most quickly after the block was invalidated (as shown in Figure 7-5), meaning that maintaining the corresponding STAB entry for a small amount of time should capture a significant number of coherence misses.

SPECweb99 does not incur as many load coherence misses as TPC-H, so there is less potential for performance improvement. Although the average dis-

tance from invalidation to subsequent load coherence miss is much longer in SPECweb99 than in TPC-H, the ECDC protocol is able to retain STAB entries for a longer period of time than in TPC-H (approximately 50,000 cycles as opposed to 5,000 cycles, as shown in Figure 7-7). Consequently, a significant fraction of false sharing misses are avoided.

For the scientific applications, coherence misses simply do not occur frequently enough for ECDC's small reduction in misses to create a significant performance benefit. With the exception of fmm, these applications are dominated by true-sharing misses. Although fmm contains a significant number of false sharing misses, the ECDC implementations are not able to eliminate these misses, indicating that before a processor is able to use a falsely shared stale data block, it usually becomes causally dependent upon a more recent operation by the processor that invalidated the block.

Of the commercial applications, the ECDC implementation is not able to improve the performance of either SPECjbb2000 or TPC-B. TPC-B is dominated by true sharing misses, and the false sharing misses that TPC-B does exhibit are not avoided by ECDC because STAB entries are deallocated soon after they are allocated (6600 cycles later, on average, for the base ECDC protocol). Although ECDC is able to eliminate a significant fraction of false sharing misses in SPECjbb2000 (30% of all false sharing misses, representing 14% of all load coherence misses), SPECjbb2000 also incurs a significant number of non-coherence misses, watering down any performance gains from a reduction in coherence misses.

The slight performance degradation that occurs in a few applications (raytrace, SPECjbb, TPC-B) is due to the infinite probe timers used for this set of data, which results in some applications polling a memory location for an excessively long period of time before the processor finally becomes causally dependent upon that write, allowing the processor to observe the new value. This is a result of imperfect critical write/polling detection. When evaluating ECDC with realistic probe timers, these slight degradations disappear, because the timers expire in a shorter period of time, as will be shown in the next section.

The merged read/write probe set configuration obtains nearly all of the performance of the base ECDC configuration, while the scalar probe set ECDC configuration sacrifices half of the performance gains for SPECweb99. Consequently, we use the merged read/write set for the remainder of our performance evaluation, because it requires fewer resources while providing similar performance compared to the full implementation.

The performance improvement obtainable via the ECDC protocol is dependent upon the length of time that each entry is able to remain in the STAB after its allocation. STAB entries are deallocated when a probe set attached to an incoming message indicates that the processor is causally dependent upon the entry's corresponding write. Figure 7-11 categorizes STAB entry deallocations by the message type that caused the deallocation: PropePropagation messages received from another processor, intolerable miss response messages, and tolerable miss response messages initiated when a stale block access has been deemed a critical write by the critical write/polling heuristics. We find that there is no single type of

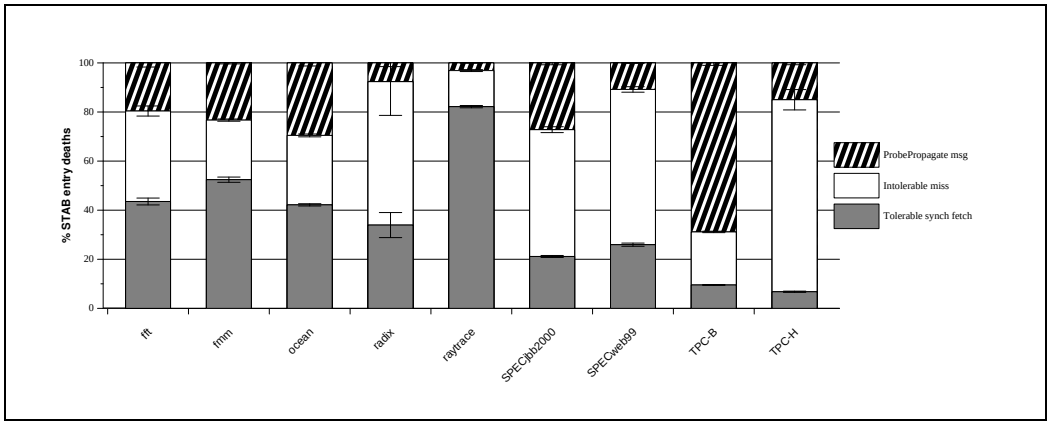

FIGURE 7-11. Causes of STAB entry deallocation.

message that dominates the removal of STAB entries across all applications; each application behaves differently. The number of STAB entry deallocations due to responses to references made by the hardware prefetcher was also measured, and was found to be small for all applications.

The data in Figure 7-11 can be used to gauge possible future improvements to the ECDC protocol. There is no potential improvement for the intolerable coherence miss category (the middle part of each bar), because these cannot be avoided; a processor must service the miss, and must consequently become causally dependent upon prior writers of that cache block. However, tolerable miss responses caused by the critical write/polling heuristics may be a source for improvement because these heuristics may not be accurate. In early evaluations of this protocol, we used a different policy for fetching new copies of stale blocks: a prefetch was initiated the first time a stale cache block was referenced, to ensure that synchronization was not delayed. However, we found that these returning prefetches consistently brought incoming probe sets that would force the removal of many probes from the STAB, accounting for the vast majority of all STAB entry deallocations. Consequently, we added the critical write/polling heuristics, which issue a coher-

ence transaction immediately for only those loads believed to be synchronization, and otherwise allow a processor to use stale data without immediately fetching the new data. Because a prefetch does not immediately fetch a new copy of the block, STAB entries live longer.

Another source of improvement may come from avoiding as many STAB deallocations due to ProbePropagation messages. The ECDC protocol uses PropePropagation messages to inform processors when they should become causally dependent upon a new probe. However, this propagation sometimes removes STAB entries prematurely, because the sending processor may never use one of the blocks that is stale in its cache. (ProbePropagation messages inform the processor that initiated the invalidation of a stale block that it should become causally dependent upon a new probe received by the processor with the stale copy, just in case the processor with the stale copy chooses to use it in the future. If the processor with the stale copy never uses it, there is no need to propagate the probe.) For some workloads such as TPC-B, a significant number of STAB entries are deallocated due to ProbePropagation messages. There may be a more intelligent mechanism for propagating probes than the mechanism used here that will mitigate this problem. We leave the exploration of these improvements to future work.

7.4 ECDC Performance Considering Resource Constraints

In the previous section, we showed that the ECDC protocol is capable of improving the performance of at least two applications from our set of benchmarks, when resource constraints were not modeled. In this section, we evaluate

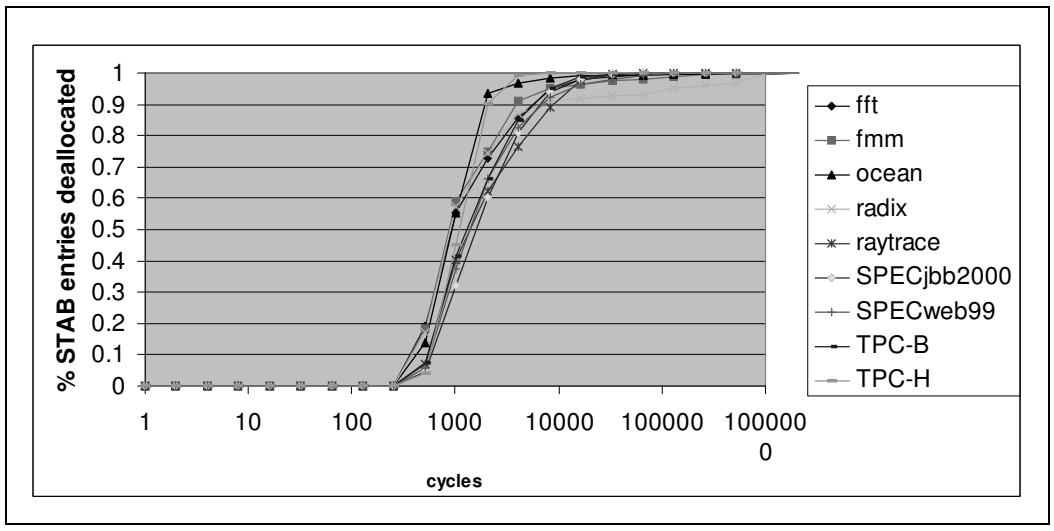

FIGURE 7-12. Useful STAB entry allocation to death distance CDF.

the effects of finite resources on the ECDC protocol. We first evaluate the use of a finite stale block lifetime timer for each probe, comparing to the infinite timer version evaluated above. We then evaluate the performance effects of finite STAB and PPB resources on this implementation, followed by a discussion of ECDC table storage requirements when assuming these finite buffers.

7.4.1 The effect of a finite stale block lifetime

Figure 7-12 illustrates the cumulative distribution function for useful STAB entry lifetime, which we use to determine the proper setting for the stale block lifetime parameter. This graph shows the number of cycles between STAB entry allocation and deallocation for those STAB entries that are useful in the ECDC protocol (whose corresponding stale block is ever used), when assuming no resource or stale block lifetime limits. For example, for the benchmark ocean, 55% of those STAB entries whose stale blocks are used are deallocated within the first 1000 cycles after the STAB entry was allocated. Across all applications, over 75% of all useful STAB entries have expired within 4k cycles. At 8k cycles, this num-

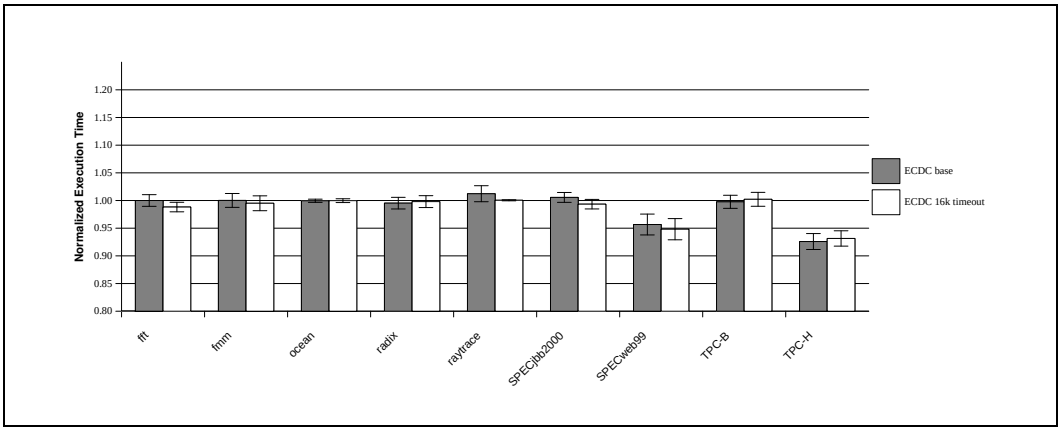

FIGURE 7-13. ECDC performance with 16k cycle stale block lifetime. Relative to the base ECDC unlimited lifetime case. Both with unlimited PPB and STAB resources.

ber increases to 89%, and at 16k, the number increases to 97%. The average STAB entry lifetime is much longer than these amounts for most applications, as shown in Figure 7-7, however the results from that figure are pulled upwards by those STAB entries that are never used. STAB entries that are useful are eliminated fairly quickly, as this data shows. Because no more than 3% of all STAB entries would be deallocated early for any application when using a 16k probe timeout, we choose this value as our probe lifetime timeout value, resulting in 14 bits of counter space per STAB entry and probe set entry.

Figure 7-13 shows the performance of the ECDC protocol relative to the baseline when using this stale block lifetime timer value. The chart also includes a bar for ECDC performance when using an infinite timer. As we can see from this data, the performance when using a 16k timer is comparable to the base ECDC performance, within the margin of error for these simulations.

In addition to performance, another important factor is the number additional messages required to send ProbePropagate notices, which is presented in Figure 7-14. This figure illustrates the number of memory barriers per 1000

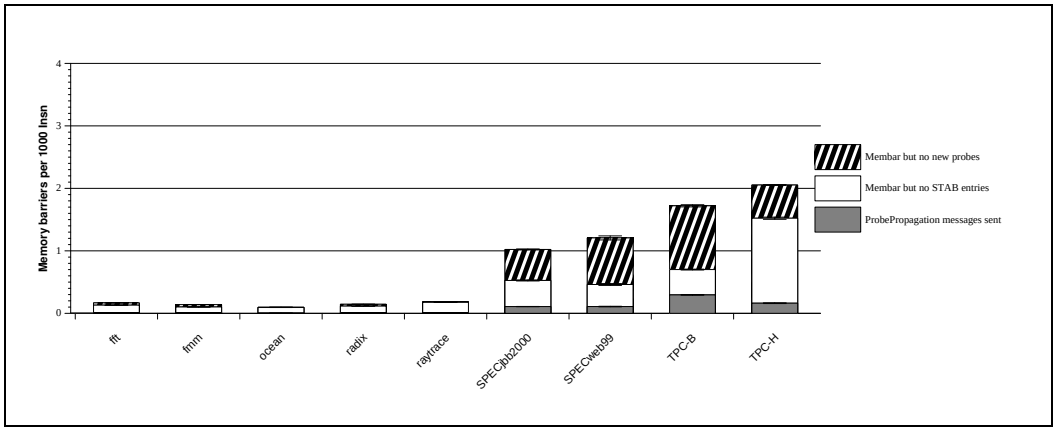

FIGURE 7-14. Measurement of required ProbePropagation messages.

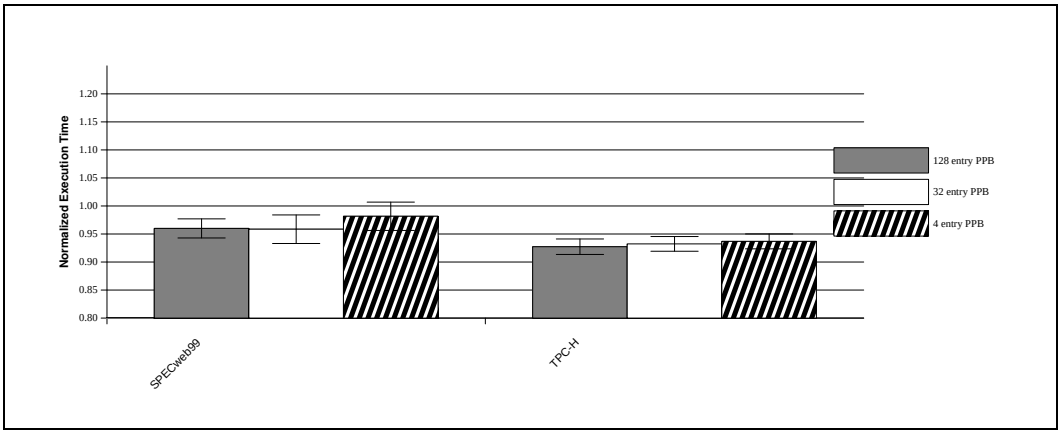

FIGURE 7-15. ECDC performance with finite STAB and PPB resources.

instructions, divided into those requiring ProbePropagation messages, and those in which the extra probe propagation message can be avoided because either the processor has no STAB entries at the time that it executes the memory barrier, or has not added a new probe to its PPB since the last time it executed a memory barrier instruction. In general, the number of additional messages is quite low for most applications. In the worst case, TPC-B, one additional message is required per 3000 instructions executed.

7.4.2 The effect of finite STAB and PPB structures

Figure 7-15 illustrates the performance of the ECDC implementation assuming finite STAB and PPB resources, for those applications that gain perfor-

mance using the ECDC protocol. This data was collected using a 128 entry STAB per processor and 32 entry CastoutPPB per directory controller. Each PPB implementation assumes a 256 virtual PPB namespace, so each timer index table entry pointer (stored in the cache with each cache block for this evaluation) consists of 8 bits for the virtual PPB name, plus log_2 *(PPB timer table entries)*. The number of PPB timer table entries is varied from 128 to 4.

As the figure illustrates, this configuration is able to capture nearly all of the benefit that was gained by the ECDC protocol when assuming infinite buffer space, within approximately 1% of the infinite STAB and PPB case for both applications. As the number of PPB timer table entries is varied, the difference in performance is within the margin of error for these applications, although the mean performance for each decreases slightly as the number of PPB timer table entries is decreased. In the next section, we present a summary of the ECDC storage capacity requirements for different ECDC configurations.

7.4.3 Analysis of ECDC storage overhead

In this section, we quantify the storage cost of the ECDC implementation used to maintain the PPB, STAB and CastoutPPB. For this data, we assume a machine with a 2 terabyte physical address space (41 bit physical addresses), which affects the size of each STAB entry. We also assume that the number of directory controllers is equal to the number of processors, resulting in one CastoutPPB per processor.

Figure 7-16 illustrates the per-processor storage space (in KB) used to implement the ECDC protocol, using the parameters that were used in the last

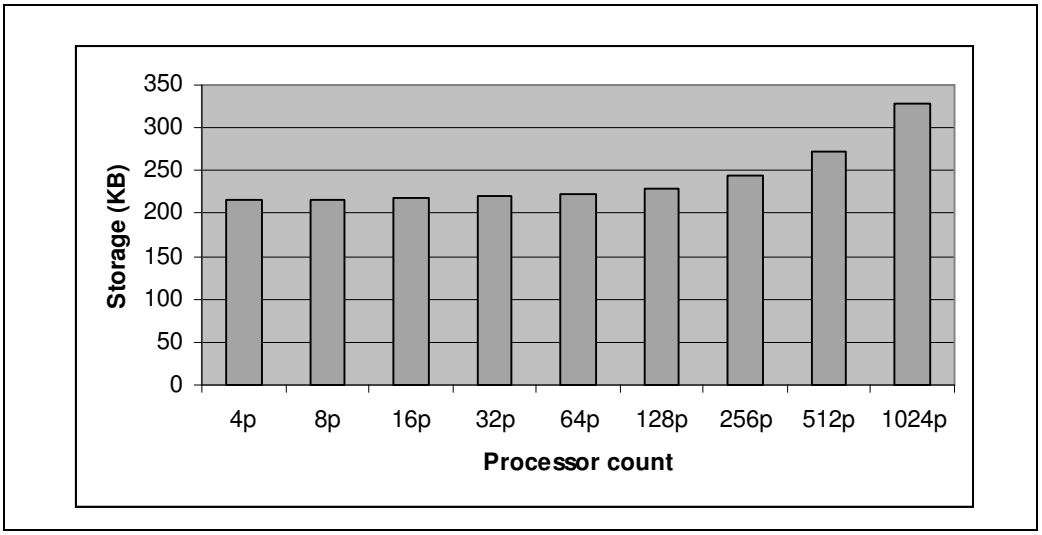

FIGURE 7-16. ECDC Storage overhead with different processor counts (probe set entries).

evaluation section (16k stale data lifetime parameter, 128 entry STAB per processor, 32 entry PPB timer table per processor, 32 entry CastoutPPB per directory controller, 256 entry virtual PPB namespace). The storage requirements are charted for systems with different processor counts, which affects the size of each PPB timer table entry. Because storage overhead is dominated by the size of the pointers stored with each cache block (13 bits per block), which is independent of the number of processors in the system, increasing processor counts have little effect on the storage requirements of the protocol until very large processor counts are used. A more significant effect of increasing processor count is the increased bandwidth consumed by probe sets attached to coherence messages. In an n processor system with a 16k stale data lifetime parameter, this becomes $14n$ bits per response message. We believe that in large systems, however, a sparse probe set representation would be more space efficient. We leave the exploration of such a sparse representation to future work.

7.5 Summary

In this chapter, we have evaluated the performance of edge-chasing delayed consistency relative to a conventional directory-based coherence protocol. There exists some opportunity for improving performance using this protocol, but not as much opportunity as was expected for the AIX applications evaluated. For these applications, STAB entries are deallocated too quickly to be very useful for most applications. Many of these deallocations occur due to probes attached to the responses to tolerable misses that were initiated due to the critical write/polling heuristics, which may be improved through better heuristics. However, many were also deallocated due to incoming probes attached to the responses of unavoidable misses. Those applications in which performance significantly improve, TPC-H and SPECweb99, receive most of their performance improvement through a reduction in false-sharing misses. These reductions result in average improved performance of 8% and 4%, for a realistic ECDC-based machine configuration running TPC-H and SPECweb99, respectively.

Although the ECDC protocol demonstrated outstanding performance gains for a lock-free linked-list manipulation microbenchmark, this programming paradigm has not yet been adopted in these AIX applications. We expect that this protocol should provide more compelling performance results in other systems in which the lock-free programming paradigm is more pervasively used, although it is too early to say for sure.

Chapter 8

Conclusions

Thanks to ever-increasing on-chip transistor capacities, shared-memory multiprocessors are becoming increasingly pervasive. Although smaller transistors may allow multiple processors to fit on a single chip, as transistor sizes decrease and clock rates increase, the relative latency of communicating with off-chip processors, or distant on-chip processors, also increases. These scaling trends also create difficulties implementing the microarchitectural structures commonly used to enforce multiprocessor consistency models. In this thesis, we have presented two solutions to these problems that take advantage of the knowledge of causal relationships among processors (or lack thereof). In the first part of the thesis, we describe and evaluate a novel memory ordering scheme called value-based replay that infers the presence of potential causal dependences in order to filter the performance-degrading effects of load replays. In the second part of the thesis, we propose and evaluate a novel delayed consistency protocol called edge-chasing delayed consistency, that allows a processor to continue to read from an invalidated cache block until the processor becomes causally dependent upon the new copy of the block, by explicitly detecting potential violations of causality using the constraint graph representation.

8.1 Value-based Replay

As transistor budgets increase, there is a great temptation to approach each

new problem by augmenting an existing design with special purpose predictors, buffers, and other hardware widgets. Such hardware may solve the problem, but also increases the size and complexity of a design, creating an uphill battle for architects, designers, and verifiers trying to produce a competitive product. We have presented a simple alternative to conventional associative load queues and shown that value-based replay causes a negligible impact on performance compared to a machine whose load queue size is unconstrained. When comparing the value-based replay implementation to processors whose load queues are constrained by clock cycle time, there is potential for significant performance benefits, up to 34% and averaging 8% relative to a 16-entry load queue, as shown in Section 5.3.

The value-based memory ordering mechanism relies on several heuristics to achieve high performance, which reduce the number of replays significantly. We believe these heuristics are more broadly applicable to other load/store queue designs and memory order checking mechanisms, and plan to explore their use in other settings. Although the evaluation presented in Chapter 5 focuses on value-based replay as a complexity-effective means for enforcing memory ordering, we believe that there is also potential for energy savings. In future work, we plan to perform a more thorough evaluation of value-based replay as a low-power alternative to conventional load queue designs.

The load queue is merely one part of a processor's microarchitecture that suffers from size constraints in conventional designs, there are many other parts of the machine that will limit the scaling of an out-of-order microprocessor's instruc-

tion window to arbitrarily large sizes. This is currently a fertile area of research, and we believe that there is much potential future work for simplifying the construction of large-window processors.

8.2 Edge-Chasing Delayed Consistency

The overheads of inter-processor communication in shared-memory multiprocessors is currently a source of much performance degradation, and will continue to be without continued research on its improvement. We have presented a new algorithm for detecting causal relationships in shared memory multiprocessors, which we have used to implement a delayed consistency mechanism that detects avoidable coherence misses. We have shown that edge-chasing delayed consistency can dramatically improve performance for lock-free list manipulation algorithms that operate on highly-contended data structures. However, the PowerPC/AIX applications studied in this thesis do not exhibit this data-race tolerant property, and consequently most of the performance gains offered by the edge-chasing delayed consistency protocol stem from its ability to tolerate false-sharing misses. We find that the ECDC protocol offers performance improvement for two of the applications studied, improving the performance of the TPC-H decision support benchmark by 8% and the performance of the SPECweb99 web serving benchmark by 4%.

Because edge-chasing delayed consistency can eliminate a significant number of false sharing misses, it would also be interesting to measure its performance as cache block size is increased beyond 128 bytes. Unfortunately, the soft-

ware running in our full system simulator livelocks when increasing cache block size beyond 128 bytes, due to an increased probability of store conditional failures caused by the increased lock granule size. Regrettably, we must leave this exploration to future work.

Although the edge-chasing delayed consistency protocol presented here offers marginal performance benefits for many applications, its ability to improve the performance of two applications illustrates that it has potential. This thesis concludes with two open questions regarding the ECDC protocol, the solutions of which may improve its performance across parallel applications. First, can probes be propagated more intelligently than the brute force ProbePropagation messages sent at memory barrier instructions? The ProbePropagation method tends to force another processor to become causally dependent upon a probe simply because the sending processor has another processor's stale block, even if that stale block will never be used. If there were a better way to detect those stale cache blocks that will not be useful in the future, many of the STAB entry deallocations could be prevented. Second, a significant percentage of STAB entries are deallocated due to incoming probes attached to predicted critical miss response messages. Although many of these deallocations are needed, because the incoming miss was correctly predicted to contain a released lock value, if our critical write heuristics are incorrect, then the miss was not necessary and the STAB entry deallocations could be avoided. The development of more accurate critical write detection heuristics may alleviate this problem.

References

[1] S. V. Adve. *Designing Memory Consistency Models for Shared-Memory Multiprocessors*. PhD thesis, University of Wisconsin-Madison, November 1993.

[2] Y. Afek, G. Brown, and M. Merritt. Lazy caching. *ACM Transactions on Programming Languages and Systems*, 15(1):182–205, January 1993.

[3] A. Ahmed, P. Conway, B. Hughes, and F. Weber. AMD Opteron shared memory multiprocessor systems. In *Proc. of the 14th HotChips Symp.*, August 2002.

[4] H. Akkary, R. Rajwar, and S. T. Srinivasan. Checkpoint processing and recovery: Towards scalable large instruction window processors. In *Proc. of the 36th Intl. Symp. on Microarchitecture*, December 2003.

[5] A. R. Alameldeen, M. M. K. Martin, C. J. Mauer, K. E. Moore, M. Xu, M. D. Hill, D. A. Wood, and D. J. Sorin. Simulating a $2M commercial server on a $2K PC. *IEEE Computer*, 36(2):50–57, 2003.

[6] A. R. Alameldeen and D. A. Wood. Variability in architectural simulations of multi-threaded workloads. In *Proc. of the Ninth Intl. Symp. on High Performance Computer Architecture*, February 2003.

[7] E. Altman, K. Ebcioglu, M. Gschwind, and S. Sathaye. Advances and future challenges in binary translation and optimization. *Proc. of the IEEE*, 89(11):1710–1722, November 2001.

[8] R. Alverson, D. Callahan, D. Cummings, B. Koblenz, A. Porterfield, and B. Smith. The Tera computer system. In *Proc. of the 1990 Intl. Conf. on Supercomputing*, pages 1–6, 1990.

[9] C. Anderson and J.-L. Baer. Design and evaluation of a subblock cache coherence protocol for bus-based multiprocessors. Technical Report UW-CSE-94-05-02, University of Washington, May 1994.

[10] T. Austin. DIVA: A reliable substrate for deep submicron microarchitecture design. In *Proc. of the 32nd Intl. Symp. on Microarchitecture*, pages 196–207, November 1999.

[11] R. Bedichek. SimNow: Fast platform simulation purely in software. In *Proc. of Hot Chips 16*, August 2004.

[12] B. N. Bershad and M. J. Zekauskas. Midway: Shared memory parallel programming with entry consistency for distributed memory multiprocessors. Technical Report CMU-CS-91-170, Carnegie Mellon University, Pittsburgh, PA, 1991.

[13] N. L. Binkert, E. G. Hallnor, and S. K. Reinhardt. Network-oriented full-system simulation using M5. In *Proc. of the Sixth Workshop on Computer Architecture Evaluation using Commercial Workloads*, February 2003.

[14] J. M. Borkenhagen, R. J. Eickemeyer, R. N. Kalla, and S. R. Kunkel. A multithreaded PowerPC processor for commercial servers. *IBM Journal of Research and Development*, 44(6):885–898, 2000.

[15] J.M. Borkenhagen, R.D. Hoover, and K.M. Valk. EXA cache/scalability controllers. IBM Enterprise X-Architecture Technology: Reaching the Summit. IBM Corporation, 2002.

[16] G. Brown. Asynchronous multicaches. *Distributed Computing*, 4(1):31–36, 1990.

[17] D. Burger, T. M. Austin, and S. Bennett. Evaluating future microprocessors: The SimpleScalar tool set. Technical Report CS-TR-1996-1308, University of Wisconsin-Madison, 1996.

[18] H. W. Cain, K. M. Lepak, and M. H. Lipasti. A dynamic binary translation approach to architectural simulation. In *Proc. of the Workshop on Binary Translation (WBT-2000)*, October 2000.

[19] H. W. Cain, K. M. Lepak, B. A. Schwartz, and M. H. Lipasti. Precise and accurate processor simulation. In *Proc. of the Workshop on Computer Architecture Evaluation using Commercial Workloads*, February 2002.

[20] H. W. Cain, M. H. Lipasti, and R. Nair. Constraint graph analysis of multithreaded programs. In *Proc. of the 12th Intl. Conf. on Parallel Applications and Compilation Techniques*, pages 4–14, September 2003.

[21] Harold W. Cain and Mikko H. Lipasti. Verifying sequential consistency using vector clocks. In *Proceedings of the 14th Symp. on Parallel Algorithms and Architectures Revue*, pages 153–154, August 2002.

[22] K. M. Chandy and J. Misra. A distributed algorithm for detecting resource deadlocks in distributed systems. In *Proc. of the Symp on Principles of Distributed Computing*, pages 157–164, August 1982.

[23] A. Charlesworth. The Sun fireplane system interconnect. In *Proc. of the 2001 Conf. on Supercomputing*, pages 7–20, 2001.

[24] A. Charlesworth, A. Phelps, R. Williams, and G. Gilbert. Gigaplane-XB: extending the ultra enterprise family. In *Proc. of Hot Interconnects V*, pages 97–112, August 1997.

[25] Y.-S. Chen and M. Dubois. Cache protocols with partial block invalidations. In *Proc. of the Seventh Intl. Parallel Processing Symp.*, pages 16–24, 1993.

[26] G. Z. Chrysos and J. S. Emer. Memory dependence prediction using store sets. In *Proc. of the 25th Intl. Symp. on Computer Architecture*, pages 142–153, 1998.

[27] W.W. Collier. *Reasoning About Parallel Architectures*. Prentice-Hall, Englewood Cliffs, NJ, 1992.

[28] Compaq Computer Corporation, Shrewsbury, Massachusetts. *21264/EV68CB and 21264/EV68DC Hardware Reference Manual*, 1.1 edition, June 2001.

[29] A. Condon and A. J. Hu. Automatable verification of sequential consistency. In *Proc. of the 13th Symp. on Parallel Algorithms and Architectures*, January 2001.

[30] F. Corella, J. M. Stone, and C. M. Barton. A formal specification of the PowerPC shared memory architecture. Technical Report 18638, IBM Research, January 1993.

[31] Z. Cvetanovic. Performance analysis of AlphaServer GS1280. White Paper, Hewlett Packard Corporation, January 2003.

[32] F. Dahlgren and P. Stenstrom. Using write caches to improve performance of cache coherence protocols in shared-memory multiprocessors. *Journal of Parallel and Distributed Computing*, 26(2):193–210, April 1995.

[33] C. Dubnicki and T. J. LeBlanc. Adjustable block size coherent caches. In *Proc. of the 19th Intl. Symp. on Computer Architecture*, pages 170–180, 1992.

[34] M. Dubois, C. Scheurich, and F. Briggs. Memory access buffering in multiprocessors. In *Proc. of the 13th Intl. Symp. on Computer Architecture*, pages 434–442, June 1986.

[35] M. Dubois, J. Skeppstedt, and P. Stenstrom. Essential misses and data traffic in coherence protocols. *Journal of Parallel and Distributed Computing*, 29(2):108–125, 1995.

[36] M. Dubois, J.-C. Wang, L. A. Barroso, K. Lee, and Y.-S. Chen. Delayed consistency and its effects on the miss rate of parallel programs. In *Supercomputing*, pages 197–206, 1991.

[37] J. Emer, P. Ahuja, E. Borch, A. Klauser, C.-K. Luk, S. Manne, S. S. Mukherjee, H. Patil, S. Wallace, N. Binkert, R. Espasa, and T. Juan. Asim: A performance model framework. *IEEE Computer*, 35(2):68–76, 2002.

[38] B. Gamsa, O. Krieger, J. Appavoo, and M. Stumm. Tornado: Maximizing lo-

cality and concurrency in a shared memory multiprocessor operating system. In *Proc. of the Third Symp. on Operating Systems' Design and Implementation*, pages 87–100, February 1999.

[39] K. Gharachorloo. *Memory Consistency Models for Shared-Memory Multiprocessors*. PhD thesis, Stanford University, 1995.

[40] K. Gharachorloo, A. Gupta, and J. Hennessy. Two techniques to enhance the performance of memory consistency models. In *Proc. of the 1991 Intl. Conf. on Parallel Processing*, pages 355–364, August 1991.

[41] K. Gharachorloo, M. Sharma, S. Steely, and S. Van Doren. Architecture and design of AlphaServer GS 320. In *Proc. of the Ninth Intl. Conf. on Architectural Support for Programming Languages and Operating Systems*, pages 13–24, November 2000.

[42] K. Goldman and K. Yelick. A unified model for shared-memory and message-passing systems. Technical Report WUCS-93-35, Department of Computer Science, Washington University, June 1993.

[43] J. R. Goodman and P. J. Woest. The Wisconsin Multicube: A new large-scale cache-coherent multiprocessor. In *Proc. of the 15th Intl. Symp. on Computer Architecture*, pages 422–431, 1988.

[44] S. Hangal, D. Vahia, C. Manovit, J.-Y. J. Lu, and S.Narayanan. TSOtool: A program for verifying memory systems using the memory consistency model. In *Proc. of the 31st Intl. Symp. on Computer Architecture*, pages 114–123, 2004.

[45] M. Herlihy. Wait-free synchronization. *Transactions on Programming Languages and Systems (TOPLAS)*, 13(1):124–149, 1991.

[46] J. Hesson, J. LeBlanc, and S. Ciavaglia. Apparatus to dynamically control the out-of-order execution of load-store instructions. United States Patent 5,615,350, March 1997.

[47] M. D. Hill. *Aspects of Cache Memory and Instruction Buffer Performance*. PhD thesis, University of California-Berkeley, 1987.

[48] J. Huh, J. Chang, D. Burger, and G. S. Sohi. Coherence decoupling: Making use of incoherence. In *Proc. of the 11th Intl. Conf. Architectural Support for Programming Languages and Operating Systems*, October 2004.

[49] Intel Corporation. *Pentium Pro Family Developers Manual, Volume 3: Operating System Writers Manual*, Jan. 1996.

[50] Intel Corporation. *Intel IA-64 Architecture Software Developers Manual, Vol-*

[51] P. Keleher, A. L. Cox, and W. Zwaenepoel. Lazy release consistency for software distributed shared memory. In *Proc. of the 19th Intl. Symp. on Computer Architecture*, pages 13–21, 1992.

[52] T. Keller, A. Maynard, R. Simpson, and P. Bohrer. SimOS-PPC full system simulator. http://www.cs.utexas.edu/users/cart/simOS.

[53] A.J. KleinOsowski and D. J. Lilja. MinneSPEC: A new SPEC benchmark workload for simulation-based computer architecture research. *Computer Architecure Letters*, 1, June 2002.

[54] E. Knapp. Deadlock detection in distributed databases. *Computing Surveys*, 19(4):303–328, 1987.

[55] S. Kunkel, B. Armstrong, and P. Vitale. System optimization for OLTP workloads. *IEEE Micro*, pages 56–64, May/June 1999.

[56] Compaq Western Research Lab. SimOS-Alpha full system simulator. http://research.compaq.com/wrl/projects/SimOS/index.html.

[57] L. Lamport. Time, clocks, and the ordering of events in a distributed system. *Communications of the ACM*, 21(7):558–565, July 1978.

[58] L. Lamport. The Wildfire verification challenge problem, June 2000. http://research.microsoft.com/users/lamport/tla/wildfire-challenge.html.

[59] A. Landin, E. Hagersten, and S. Haridi. Race-free interconnection networks and multiprocessor consistency. In *Proc. of the 18th Intl. Symp. on Comp. Architecture*, 1991.

[60] J. Laudon and D. Lenoski. The SGI Origin: a ccNUMA highly scalable server. In *Proc. of the 24th Intl. Symp. on Computer Architecture*, pages 241–251, 1997.

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

[62] D. Lenoski, J. Laudon, K. Gharachorloo, A. Gupta, and J. Hennessy. The directory-based cache coherence protocol for the DASH multiprocessor. In *Proc. of the 17th Intl. Symp. on Computer Architecture*, pages 148–159, 1990.

[63] K. M. Lepak. *Exploring, Defining, and Exploiting Recent Store Value Locality*. PhD thesis, University of Wisconsin-Madison, 2003.

[64] K. M. Lepak, H. W. Cain, and M. H. Lipasti. Redeeming IPC as a performance

metric for multithreaded programs. In *Proc. of the 12th Intl. Conf. on Parallel Architectures and Compilation Techniques*, pages 232–243, 2003.

[65] K. M. Lepak and M. H. Lipasti. On the value locality of store instructions. In *Proc. of the 27th Intl. Symp. on Computer Architecture*, pages 182–191, Vancouver, BC, June 2000.

[66] K. M. Lepak and M. H. Lipasti. Temporally silent stores. In *Proc. of the 10th Intl. Conf. on Architectural Support for Programming Languages and Operating Systems*, San Jose, California, October 2002.

[67] M. H. Lipasti and J. P. Shen. Exceeding the dataflow limit via value prediction. In *Proc. of the 29th Intl. Symp. on Microarchitecture*, pages 226–237, 1996.

[68] B. Liskov and R. Ladin. Highly available distributed services and fault-tolerant distributed garbage collection. In *Proc. of the Fifth Symp. on Principles of Distributed Computing*, pages 29–39, 1986.

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

[70] M. M. K. Martin, D. J. Sorin, A. Ailamaki, A. R. Alameldeen, R. M. Dickson, C. J. Mauer, K. E. Moore, M. Plakal, M. D. Hill, and D. A. Wood. Timestamp snooping: an approach for extending smps. In *Proc. of Ninth Intl. Conf. on Architectural Support for Programming Languages and Operating Systems*, pages 25–36, November 2000.

[71] 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 *Proc. of the 34th Intl. Symp. on Microarchitecture*, pages 328–337, December 2001.

[72] C. J. Mauer, M. D. Hill, and D. A. Wood. Full-system timing-first simulation. In *Proc. of the 2002 SIGMETRICS Intl. Conf. on Measurement and modeling of computer systems*, pages 108–116, 2002.

[73] C. May, E. Silha, R. Simpson, and H. Warren, editors. *The PowerPC Architecture: A Specification for a New Family of RISC Processors, 2nd edition*. Morgan Kauffman, San Francisco, California, 1994.

[74] A. M. G. Maynard, C. M. Donnelly, and B. R. Olszewski. Contrasting characteristics and cache performance of technical and multi-user commercial workloads. In *Proc. of the Sixth Intl. Conf. on Architectural Support for Programming Languages and Operating Systems*, pages 145–156, 1994.

[75] P. E. McKenney. *Exploiting Deferred Destruction: An Analysis of Read-Copy-Update Techniques in Operating System Kernels*. PhD thesis, OGI School of Science and Engineering at Oregon Health and Sciences University, 2004.

[76] P. E. McKenney and J. D. Slingwine. Read-copy update: Using execution history to solve concurrency problems. In *Parallel and Distributed Computing and Systems*, pages 509–518, Las Vegas, NV, October 1998.

[77] M. M. Michael. High performance dynamic lock-free hash tables and list-based sets. In *Proc. of the 14th Symp. on Parallel Algorithms and Architectures*, pages 73–82, August 2002.

[78] M. M. Michael. Safe memory reclamation for dynamic lock-free objects using atomic reads and writes. In *Proc. of the 21st Symp.on Principles of Distributed Computing*, pages 21–30, 2002.

[79] M. M. Michael. Scalable lock-free dynamic memory allocation. In *Proc. of the 2004 Conf. on Programming Language Design and Implementation*, pages 35–46, 2004.

[80] G.E. Moore. Cramming more components onto digital circuits. *Electronics*, 38(8):114–117, 1965.

[81] A. Moshovos, S. E. Breach, T. N. Vijaykumar, and G. S. Sohi. Dynamic speculation and synchronization of data dependences. In *Proc. of the 24th Intl. Symp. on Computer Architecture*, pages 181–193, 1997.

[82] M. Moudgill, J-D. Wellman, and J. Moreno. Environment for PowerPC microarchitecture exploration. *IEEE Micro*, 19(3):15–25, May/June 1999.

[83] T. Mudge. Power: A first-class architectural design constraint. *IEEE Computer*, 34(4):52–58, April 2001.

[84] S. S. Mukherjee, P. Bannon, S. Lang, A. Spink, and D. Webb. The Alpha 21364 network architecture. *IEEE Micro*, 22(1):26–35, 2002.

[85] S. Onder and R. Gupta. Dynamic memory disambiguation in the presence of out-of-order store issuing. In *Proc. of the 32nd Intl. Symp. on Microarchitecture*, pages 170–176, 1999.

[86] V. S. Pai, P. Ranganathan, and S. V. Adve. RSIM: An execution-driven simulator for ILP-based shared-memory multiprocessors and uniprocessors. In *Proc. of the Third Workshop on Computer Architecture Education*, February 1997.

[87] Il Park, C.-L. Ooi, and T. N. Vijaykumar. Reducing design complexity of the load-store queue. In *Proc. of the 36th Intl. Symp. on Microarchitecture*, Decem-

[88] D. S. Parker, G. J. Popek, G. Rudisin, A. Stoughton, B. Walker, E. Walton, J. Chow, D. Edwards, S. Kiser, and C. Kline. Detection of mutual inconsistency in distributed systems. *IEEE Transactions on Software Engineering*, 9(3):240–247, May 1983.

[89] D. Ponomarev, G. Kucuk, and K. Ghose. Reducing power requirements of instruction scheduling through dynamic allocation of multiple datapath resources. In *Proc. of the 34th Intl. Symp. on Microarchitecture*, December 2001.

[90] S. Qadeer. On the verification of memory models of shared-memory multiprocessors. In *Proc. of the 12th Intl. Conf. on Computer Aided Verification*, 2000.

[91] R. Rajwar. *Speculation-Based Techniques for Transactional Lock-Free Execution of Lock-Based Programs*. PhD thesis, University of Wisconsin-Madison, 2002.

[92] R. Rajwar, A. Kagi, and J. R. Goodman. Improving the throughput of synchronization by insertion of delays. In *Proc. of the Sixth Intl. Symp. on High-Performance Computer Architecture*, January 2000.

[93] R. N. Rechtschaffen and K. Ekanadham. Multi-processor cache coherency protocol allowing asynchronous modification of cache data. United States Patent 5,787,477, July 1998.

[94] P.F. Reynolds, Jr., C. Williams, and R. R. Wagner, Jr. Isotach networks. *IEEE Transactions on Parallel and Distributed Systems*, 8(4):337–348, 1997.

[95] M. Rosenblum, E. Bugnion, S. A. Herrod, E. Witchel, and A. Gupta. The impact of architectural trends on operating system performance. In *Proc. of the fifteenth Symp. on Operating systems principles*, pages 285–298, 1995.

[96] M. Rosenblum, S. Herrod, E. Witchel, and A. Gupta. Complete computer simulation: the SimOS approach. *IEEE Parallel and Distributed Technology*, 3(4):34–43, 1995.

[97] S. Sethumadhavan, R. Desikan, D. Burger, C. R. Moore, and S.W. Keckler. Scalable hardware memory disambiguation for high-ILP processors. In *Proc. of the 36th Intl. Symp. on Microarchitecture*, December 2003.

[98] D. Shasha and M. Snir. Efficient and correct execution of parallel programs that share memory. *Transactions on Programming Languages and Systems*, 10(2):282–312, April 1988.

[99] P. Shivakumar and N. P. Jouppi. Cacti 3.0: An integrated cache timing, power,

and area model. Technical Report 2001/2, Compaq Western Research Lab Research Report, 2001.

[100] P. S. Sindhu, J.-M. Frailong, and M. Cekleov. Formal specification of memory models. Technical Report CSL-91-11, Xerox Corporation, December 1991.

[101] R. L. Sites, editor. *Alpha Architecture Reference Manual*. Digital Press, Maynard, MA, 1992.

[102] IEEE Computer Society. *IEEE Standard for Scalable Coherent Interface (SCI)*. Washington, DC, 1993.

[103] G. S. Sohi and S. Vajapeyam. Instruction issue logic for high-performance, interruptable pipelined processors. In *Proc. of the 14th Intl. Symp. on Computer Architecture*, pages 27–34, 1987.

[104] D.J. Sorin, M. M. K.. Martin, M. D. Hill, and D.A. Wood. Safetynet: improving the availability of shared memory multiprocessors with global checkpoint/recovery. In *Proc. of the 29th Intl. Symp. on Computer Architecture*, pages 123–134, 2002.

[105] R. Strom and S. Yemini. Optimistic recovery in distributed systems. *Transactions on Computer Systems*, 3(3):204–226, 1985.

[106] S. V. Tambat and S. Vajapeyam. Non-strict cache coherence: Exploiting data-race tolerance in emerging applications. In *Proc. of the 2000 Intl. Conf. on Parallel Processing*, pages 87–94, August 2000.

[107] J. M. Tendler, S. Dodson, S. Fields, H. Le, and B. Sinharoy. POWER4 system microarchitecture. Technical white paper, IBM Server Group, October 2001.

[108] S. Thompson, P. Packan, and M. Bohr. MOS scaling: Transistor challenges for the 21st century. *Intel Technology Journal*, 2(3), 1998.

[109] P. Wang and W. E. Weihl. Scalable concurrent b-trees using multi-version memory. *Journal of Parallel and Distributed Computing*, 32(1):28–48, 1996.

[110] D. L. Weaver and T. Germond, editors. *SPARC Architecture Manual (Version 9)*. PTR Prentice Hall, 1994.

[111] S. Woo, M. Ohara, E. Torrie, J. Singh, and A. Gupta. The SPLASH2 programs: Characterization and methodological considerations. In *Proc. of the 22nd Intl. Symp. on Computer Architecture*, pages 24–36, June 1995.

[112] Min Xu, Rastislav Bodik, and Mark D. Hill. A "flight data recorder" for enabling full-system multiprocessor deterministic replay. In *Proc. of the 30th Intl.*

Symp. on Computer architecture, pages 122–135, 2003.

[113] K. C. Yeager. The MIPS R10000 superscalar microprocessor. *IEEE Micro*, 16(2):28–40, April 1996.

[114] A. Yoaz, M. Erez, R. Ronen, and S. Jourdan. Speculation techniques for improving load related instruction scheduling. In *Proc. of the 26th Intl. Symp. on Computer Architecture*, pages 42–53, 1999.

[115] A. Yoaz, R. Ronen, R. S. Chappell, and Y. Almog. Silence is golden? In *Work-in-progress workshop of the 7th Intl. Symp. on High-Performance Computer Architecture*, January 2001.