ACOHERENT SHARED MEMORY

by

Derek R. Hower

A dissertation submitted in partial fulfillment of

the requirements for the degree of

Doctor of Philosophy

(Computer Sciences)

at the

UNIVERSITY OF WISCONSIN-MADISON

2012

Date of final oral examination: 7/16/2012

The dissertation is approved by the following members of the Final Oral Committee:

Ben Liblit, Associate Professor, Computer Sciences
Mikko H. Lipasti, Professor, Electrical and Computer Engineering
Michael M. Swift, Assistant Professor, Computer Sciences
David A. Wood, Professor, Computer Sciences
Mark D. Hill, Professor, Computer Sciences (Advisor)

Abstract

The computer industry has entered an era where energy and efficiency are first-class design constraints. Architects can no longer keep all of the components on a die powered simultaneously, and so there is mounting pressure to find new ways in which to use the silicon available. We think that coherence in particular is prime candidate for rethinking because of its well-known drawbacks and the increasing importance of the memory system in large multiprocessors and heterogeneous designs.

In order to find an efficient alternative to existing coherence mechanisms while still maintaining shared memory, we believe that hardware must be able to exploit the semantic information available in software. Programmers classify and group data based on usage, such as thread-private or read-only, yet existing coherence mechanisms treat all data the same. As a result, coherence hardware often wastes resources tracking state (e.g., in a directory) to allow transitions that data semantics dictate will never happen. Even when data is shared, it is usually synchronized among threads and communicated at a coarse granularity. Despite this, coherence continues to track blocks at a fine granularity as if they were all truly independent.

In this dissertation we explore Acoherent Shared Memory (ASM), a new model that facilitates the transfer of semantic information from software to hardware. ASM expresses the diversity of data semantics in a program by allowing software to group data based on use, e.g., by grouping together thread-private variables. ASM also allows software to communicate the semantics of synchronized data using an acoherent abstraction of memory. Acoherence takes its inspiration from software revision control systems like CVS, and exposes checkout and checkin operations to software.

We show that the ASM model can lead to efficient implementations by designing and evaluating ASM-CMP, an initial prototype for single-chip multiprocessors. ASM-CMP performs on par with comparable coherent systems while requiring less area (e.g., 10x less cache overhead), less energy (e.g., 18% less memory system energy), and simpler hardware. In ASM-CMP, cache controllers make entirely local decisions, and avoid the complexities that arise from distributed race conditions in conventional cache coherence protocols.

Acknowledgements

A dissertation is supposed to be a reflection of *personal* mastery of a subject as well as an *individual* contribution to the state of the art. Nevertheless, it would be foolish to believe that any accomplishment of this magnitude is the result of a single person's effort. I would not have been able to reach this point without the encouragement and support of family, friends, and colleagues.

First, I must thank my wife, Christina, who constantly reminds me of life's true priorities. She has made multiple personal sacrifices for me to reach this point, and for that I am and always will be eternally grateful. Marrying her is the smartest thing I have ever done. A Turing Award would pale in comparison.

I would never have reached grad school, much less completed it, without the support of my family. From a young age they encouraged the curiosity that has led me to become a researcher. My parents taught me the value of hard work and determination and always made me believe that I could do anything I set my mind to.

Graduate school is defined, more so than any other component, by the relationship between student and advisor. I have been extremely blessed to find an advisor who understands with great mastery both the intellectual and personal sides of graduate study. Mark's success, and the success of my elder academic siblings, is due in no small part to his ability to balance the demands of cutting edge research with the realities of life. I find myself frequently repeating his "Markisms" to younger students as I realize more and more how profound they really are. To me, Mark is the pinnacle of a scholar, and I know that I will spend the rest of my career trying to emulate his example.

Other Wisconsin faculty have also helped me along the way. David Wood, my academic godfather, has always been a tremendous resource. His ability to deduce the crux of any problem never fails to amaze me. Mike Swift, who is perhaps quite literally a walking encyclopedia, has provided great feedback and direction. Ben Liblit inspired me early on to look outside the sometimes narrow box of architecture. Guri Sohi, Karu Sankaralingam, and Mikko Lipasti have also provided feedback over the years.

I thank Dan Sorin for fostering my interest in research. Despite joining his group as a wide-eyed and naive undergraduate, he treated me as a colleague from day one. His belief in my abilities and willingness to let me wander intellectually allowed me to catch the research bug early on. Dan taught me that engineering is not black and white, and it is the shades of grey that continue to make research interesting.

My experience in graduate school has been greatly enhanced by my peers in the Wisconsin architecture group. I often looked (and continue to look) to those more senior than I for advice and direction. These include Matt Allen, Jichuan Chang, Natalie Enright-Jerger, “sensei” Mike Marty, Kevin Moore, and Phillip Wells. Their tutelage has been invaluable, and I have tried to pay it forward with students my junior.

I must thank those more contemporary with my path for countless impromptu discussions and commiserating. These include Arka Basu, Spyros Blanas, Emily Blem, Jayaram Bobba, Marc DeKruijf, Polina Dudnik, Yasuko (Watanabe) Eckert, Dan Gibson, Venkat Govindaraju, Gagan Gupta, Joel Hestness, Jason Power, Lena Olson, Marc Orr, Jason Power, Somayeh Sardashti, Rathijit Sen, Dana Vantrease, Haris Volos, James Wang, and Luke Yen. I would especially like to thank my office mates over the years, whose close proximity often made them prime candidates for random interruptions: Jichuan Chang, Oscar Plata, James Wang, Rathijit Sen, and

Jayneel Gandhi. My life has also been enhanced with great personal friendships with many of these students. They know who they are.

I have received great advice over the years from the Wisconsin Architecture Affiliates. I must especially thank Brad Beckmann, John Davis, Konrad Lai, Kevin Moore, Steve Reinhardt, and Dana Vantrease for taking a personal interest in my work and career.

My interest in computers in general was started well before graduate school. My companionship with close personal friends Cameron Cooper and Curtis Wilson, who shared a fascination with computers, no doubt fueled my early excitement. I credit my interest in computer architecture in particular to John Board at Duke, whose captivating courses and ability to make sense of complexity showed me beauty of hardware design.

Table of Contents

List of Tables

List of Figures

1

Introduction

Hardware treats all memory the same. Software does not. Commodity multicore processors use the same cache coherence protocol to manage all shared data, and assume that all data is shared at a fine granularity. In reality, software differentiates data based on semantic context, distinguishing, for example, between data that is thread-private, read-only, or synchronized. Despite this, hardware designs continue to pretend that all data is equal and spend considerable resources to support fine-grained interactions that semantics dictate will never occur. Now that energy and efficiency are first-class hardware design constraints, the time is ripe to rethink the memory system design.

1.1 Rethinking Coherence

Cache coherence has been present in commodity multiprocessor systems since at least the Intel 80386 in 1985 [94]. It has remained ubiquitous despite many well-known drawbacks, including high design complexity [40,160], large area overheads [62,183,185], and inefficient policies [1,162,167]. Architects have continued to support coherence because it has some programming benefits compared to alternatives like message passing [160] and because there are decades worth of legacy codes that expect it. But now times are changing.

1.1.1 New Constraints

Energy matters. Complexity matters. Area matters. Compatibility is flexible. What were borderline ludicrous statements in the late 20th century are now accurate depictions of reality in the 21st [110,145,163,189]. After years of unabated technology progress that gave computer architects more transistors effectively for free (see "The Prophets Speak" on page 3), processor designs evolved over time with a general disregard for energy and/or area efficiency. Performance came first, compatibility was a must, and all other constraints took a distant second. As a result many designs feature components, like coherence, that have notable inefficiencies.

Energy Matters. No exponential growth can last forever. We are now in an era of computer design where energy, area, and complexity are first-order constraints. Processors are now so power constrained that not all transistors can be turned on at the same time [38,58]. The percentage of this dark silicon, as it has been coined, is only projected to grow. A seminal study by Esmaeilzadeh et al. predicts that by the time we reach 8nm (~2020), more than half of a chip will have to be turned off to stay under the power budget [58]. In order to keep the most important parts of a chip running, we predict that future designs will reconsider legacy support.

The Prophets Speak: How to Drive Exponential Growth

Historically, two truisms fueled the unparalleled growth of the semiconductor industry through the 20th century. The first, known as Moore's Law, was a prediction (perhaps more accurately called a mandate since the namesake, Gordon Moore, was a founder of the Intel Corporation) that that the number of devices per unit area would double approximately every year [135]. The second, known as Dennard Scaling, predicted that the per-device power and speed would scale down commensurate with the feature size, meaning that power density of a die would effectively stay constant even as more transistors are packed into the same area [51]. Combined, these two truisms afforded architects the truly remarkable property that with each new technology generation the number of devices doubled essentially "for free" [163]. Under such ideal constraints, there was little incentive to optimize energy or area efficiency in processors.

Moore's Law continues to this day, though on a revised timescale where density doubles every 18-24 months [136]. Around the turn of the 21st century, though, ideal Dennard Scaling stopped. Manufactures can still pack more transistors per unit area with each new generation, but it now takes more energy to keep that unit area running. As a result, energy is now a first-class design constraint and design efficiency is of utmost importance.

Complexity Matters. Energy constraints are also driving architects to consider heterogeneous designs that feature specialized execution units [44,87,89]. Specialized hardware can perform tasks with an order of magnitude less energy compared to equivalent software on a general purpose computation unit, thereby reducing energy overall [44]. The units can also be turned off when not in use, and are thus a viable way to handle the dark silicon problem. However, because specialized units require more custom hardware design, managing and reducing complexity, especially in interfacing components like coherence, will be a key enabler going forward.

Area Matters. In addition to the energy problems, there are also economic motivators that will change the way future hardware is designed. Historically, the cost per transistor on a die has fallen commensurate with device size in new technology generation. This trend has allowed manufacturers to pack more transistors on die while maintaining constant prices from generation to generation. Recent data from Nvidia shows that trend may be stopping [91]. As a result, it is likely manufacturers may choose not to design their products in new technology generations, instead opting for older and cheaper devices. Of course, the pressure to increase value in new products will not stop, and so designers will have to come up with better ways to use unchanging transistor budgets. In other words, area matters.

Compatibility is Flexible. In part because of the above problems, and in part because computing has migrated from the middle ground of desktops and servers to the extremes of mobile and datacenter platforms, we are seeing more and more companies evolve to become vertically integrated. Companies such as Apple, Intel, and Microsoft all make mobile devices in which they have designed both the hardware and system software stack. Other companies, such as Oracle, design the entire stack of server platforms. Because of this integration, backward compatibility is no longer a nonnegotiable design constraint. Companies have the flexibility to design hardware to fit their needs without having to worry about the effect on customers, and there is evidence that this is already occurring [110].

1.1.2 Coherence is Ripe for Change

Cache coherence protocols are notoriously complex [40,126], consume significant chip area (e.g., 14% of cache area [161]), and drain energy (we find 20% of on-chip memory energy). Most of this overhead is wasted. Previous characterizations have shown that embarrassingly parallel workloads share surprisingly little data (e.g., < 1.5% of reads and < 20% of writes).

Other, more economically relevant workloads share even less [16,61]. As we move into the era of big data, it feasible that many parallel workloads will not use it all, opting instead for paradigms like message passing.

Coherence is even wasteful for shared data. The majority of shared data is protected with synchronization, such that a single event (e.g., an unlock) signals an intent to share an entire region of data. Rather than using this information to amortize the cost of data management, coherence instead redundantly tracks the information for all of the blocks within a single synchronized region. When a region is shared, a protocol performs the same transition sequence for every involved block despite the fact that a single handshake for the region would suffice.

Coherence is also notoriously complex to implement. In directory protocols, which now dominate the market because of their good scaling properties, distributed races between nodes can cause a state space explosion [52]. This explosion leads to complexity that is difficult to reason about and even more difficult to verify correct [52,55,184]. This complexity can raise the barrier to entry for future heterogeneous components that interact with shared memory [100] and raise already high design costs.

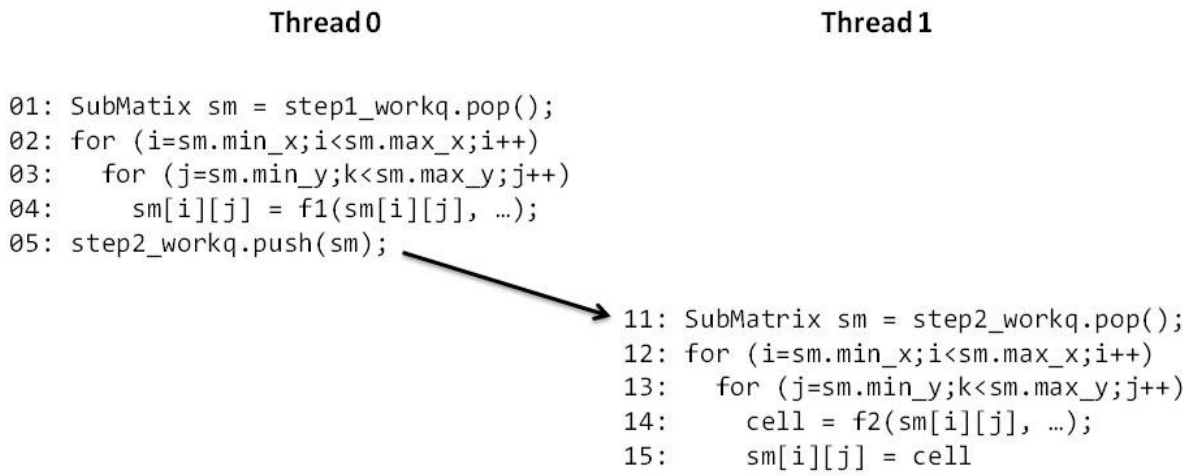

Figure 1-1. Example of synchronized data in an excerpt from a matrix algorithm.

1.2 How Data is Shared

The majority of shared data in an application is protected by synchronization constructs. In fact, in most programs the only unprotected data are the synchronization variables themselves. Synchronized data has four relevant semantic points, which we show in Figure 1-1:

Write. (line 04) The entire submatrix is updated with a sequence of instructions.
Publish. (line 05) All of the writes on line 04 become visible with synchronization.
Subscribe. (line 11) Thread 1 waits for the data to become visible.
Read. (line 14) Thread 1 consumes all of the updates from line 04.

The coherent shared memory model can only represent two of these points, namely write and read via normal stores and loads. As a result, protocols are forced to make assumptions about data that often are not true. For example, many invalidation-based protocols will incorrectly assume the read on line 14 will not be followed by a write, leading to an unnecessary coherence upgrade when the write on line 15 issues [162]. Similarly, the assumption made by most protocols that updates must be shared immediately leads to issues related to false sharing [167].

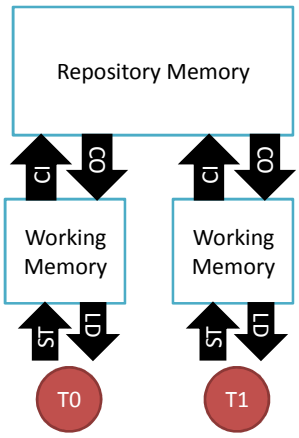

Figure 1-2. The acoherent memory abstraction.

1.3 Acoherent Shared Memory

In this dissertation we explore a new shared memory model called *Acoherent Shared Memory (ASM)*. ASM aims to address the issues with conventional shared memory stated above in two ways. First, ASM uses *semantic segmentation* as a way for software to communicate the semantics of individual data to hardware. In ASM, software can, for example, classify a variable as thread-private by allocating it from a special segment reserved for private data.

Second, ASM introduces a new abstraction of shared memory called *acoherence* that gives software the ability to convey the semantic points of synchronized data to hardware. The acoherent abstraction, shown in Figure 1-2, is modeled after software revision control systems like CVS [37] that have proven to be highly successful models for managing concurrency. In the acoherent model, software threads see a thread private working memory that they can access using normal loads and stores. Threads communicate using *checkout* and *checkin* operations that transfer updates between their private working memory and a globally shared repository memory.

Through the course of this dissertation we show that the ASM model is easy for software to use and can lead to high-performance, low energy, and low complexity memory system implementations. Intuitively, the model is simple for programmers because it is an expression of what software already does. Software already, at an algorithmic level, differentiates data based on usage. And most parallel software uses well defined synchronization, usually in the form of a library, to indicate the semantic points of synchronized data. In fact, we will show that many parallel application codes written for conventional shared memory will continue to work correctly in an ASM system without any source code changes.

We describe an initial prototype for a multicore machine called ASM-CMP. It performs on par with conventional cache coherent systems and does so without needing broadcast communication, a large coherence directory, or a complex distributed state machine (i.e., a coherence protocol) that are often found in commodity multicores. In ASM-CMP, decisions made at cache controllers are completely local (only consider processor-side requests), and thus avoid the state explosion and complexity that comes with distributed, racing requests.

We evaluate ASM-CMP compared to conventional cache coherent systems and show that ASM-CMP performs on par with conventional systems while requiring less area (e.g., 10x state overhead reduction) and less energy (20% less on-chip memory energy). From this evaluation we conclude that the ASM-CMP design, and the ASM model in general, is a plausible solution to the problems posed by existing coherent shared memory implementations.

1.4 Contributions

In our view, this dissertation makes the following contributions to the state of the art in computer architecture research:

- We recognize the need for a model of shared memory that can exploit the semantic characteristics of data, and propose the ASM model as a viable approach. ASM builds on the coherent shared memory model in two ways. First, it provides a means for software to label data based on semantic type so that hardware can apply smart management policies. Second, ASM uses the novel acoherent abstraction of memory that gives software the ability to communicate the semantic points of an important class of shared data that is protected with synchronization.
- We show that the ASM model is easy for programmers to reason about. We find many existing shared memory applications will work without modification by linking with ASM-aware libraries.
- We formally describe and rigorously analyze the ASM model. Our formalizations lend many insights into the model that are relevant to both programmers who write software and architects who design ASM hardware.
- We develop and evaluate an ASM design for a single-chip multiprocessor system. We show that by using semantic information, the design can perform on par with existing cache coherent systems while requiring less area, less energy, and a simpler cache controller.

1.5 Dissertation Structure

This dissertation is organized as follows. In Chapter 2 we describe the ASM model informally and show, through four case studies, how programmers can use the model to write software. In Chapter 3 we formalize ASM consistency. We show that ASM, despite being a different abstraction than flat, coherent memory, is still quite similar to existing and highly used consistency models like Total Store Order [176]. We also prove that ASM is equivalent to sequential consistency for data-race-free programs. Chapter 4 explores the design of the ASM-CMP prototype, showing how it implements both semantic segmentation and the acoherent memory abstraction. In Chapter 5 we outline the methods used to evaluate ASM-CMP as well as a description of the workloads used. We present an evaluation of ASM-CMP compared to two

cache coherent baselines in Chapter 6 and show that it performs comparably while requiring less energy, less area, and simpler hardware. Finally, in Chapter 7 we conclude and propose future work. In particular, we suggest ways that the ASM model might be improved, how the ASM-CMP implementation could be optimized, and how the contributions and lessons from this dissertation can be applied to more conventional systems.

2

Acoherent Shared Memory

In order to overcome the shortcomings of conventional shared memory discussed in Chapter 1, two aspects need modification. First, there is a fundamental disconnect between the way that current shared memory abstractions present memory and the way that synchronized data uses it. Synchronized data has four points of interest: the *write* point, at which software writes a new value to the location, the *publish* point, at which the updated data semantically becomes visible to other threads in the system, the *subscribe* point, where a remote thread announces its desire to see published data, and the *read* point, where the remote thread actually consumes new data. Current shared memory abstractions are only capable of representing the write and read points via normal loads and stores.

Second, there is no way to reflect the diverse range of data semantics that exist *simultaneously* in shared memory applications. In current designs, all data is treated equally regardless of the fact that different data locations are accessed in very different ways. Without a method to convey

the semantic information to hardware, shared memory implementations will continue to make poor data management decisions for a significant fraction of memory.

The Acoherent Shared Memory (ASM) model solves both problems. First, ASM introduces a new abstraction of shared memory called *acoherence* that matches the semantics of synchronized data. Second, ASM introduces the notion of *semantic segments* as a mechanism to convey access characteristics of individual datum from software to hardware.

In the rest of this chapter we will explain acoherence and semantic segmentation in more detail, and show how each can be used by software to convey useful information to hardware. In Section 2.1 we provide a detailed overview of the acoherent abstraction of shared memory. Then in Section 2.2 we discuss how semantic segments work and how they can be used to "color" memory in a way that makes fine-grained semantic differentiation easy for both software and hardware. We discuss the implications of the ASM model on system-level software in Section 2.3. Then, in Section 2.4 we provide intuition on how software might use the ASM model for various generic usage patterns. We then build on that high level foundation in Section 2.5 by exploring four detailed case studies that display four different ways to write ASM software. Finally, in Section 2.6 we compare ASM to other memory models by surveying related work.

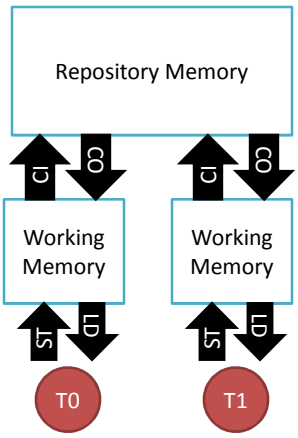

Figure 2-1. The acoherent memory abstraction.

2.1 Acoherence

Acoherence is a model of shared memory that is neither coherent (multiple valid copies of data can exist at the same time) nor incoherent (updates are shared when it matters). The acoherent view of memory, shown in Figure 2-1, resembles the abstraction provided by software revision control systems like CVS [37] or Subversion [190]. Each thread is given a local *working memory* and all threads have shared access to a global *repository memory*. All memories in the system represent the same address space[1]; address A is address A no matter where it is located. All local reads and writes operate on local memory, and cannot directly affect repository memory. Threads can access the data in repository memory using special **`checkout`** and **`checkin`** instructions. A **`checkout`** (logically) copies the entire contents of repository memory into local memory, and **`checkin`** pushes all local updates performed since the last checkout to repository memory.

[1] For now, the reader may assume the address space is equivalent to the entire addressable memory in an application. In Section 2.2 we show how this assumption can be relaxed so that acoherent operations can be applied at a finer granularity.

Acoherence transforms sharing from a two- to a four- phase operation, in line with the actual semantics of synchronized data. Data *writes* are performed with normal store operations to local memory. A thread later *publishes* that update with a **`checkin`** operation. A remote thread *subscribes* to the write with a checkout, and finally *reads* the update with a normal load. Note that like synchronized data, a single **`checkout`** or **`checkin`** operation can be used to publish or subscribe to multiple individual writes or reads.

Acoherence increases software's responsibility by making sharing a four-phase operation. As we will argue throughout the remainder of this chapter, though, this extra responsibility is a small imposition and can even be beneficial to software writers. The imposition is small because, by design, the acoherent model is a representation of what software already does; it simply makes an implicit action (synchronized sharing) explicit. And, as we will show in Section 2.5, the explicit action can often be subsumed by an operation at a higher level of abstraction like a lock, effectively hiding the imposition from application software. The acoherent requirements can even be helpful to software developers by forcing them to *think* about when sharing occurs in a program, potentially heading off performance bugs early in the design stage.

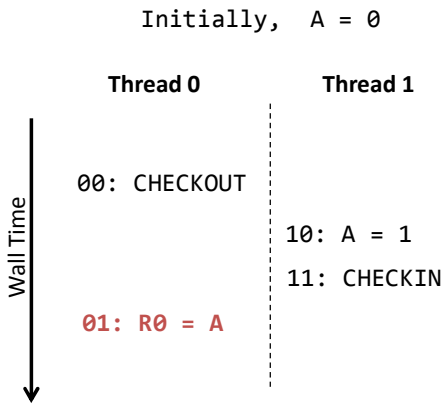

Figure 2-2. Lazy checkout

2.1.1 Subtleties

While the CVS analogy is a good first order approximation of acoherence, there are subtle differences that make the comparison inexact. First, the acoherent model does not guarantee that a write from another thread will remain isolated until a checkout is performed. Rather, it only guarantees that a checked in write will be seen after a **`checkout`**. This is because acoherence uses lazy checkouts; rather than taking an instant snapshot of repository memory, the **`checkout`** operation is free to delay performing the working memory copy until a location is observed with a read or write. We show this distinction graphically in Figure 2-2. Register **`R0`** can observe either the value zero or the value one depending on when the **`checkout`** operation on line 00 makes a copy of **`A`**.[2] We give a more formal treatment of this subtlety in Chapter 3.

In another departure from the CVS model, acoherence only provides one conflict resolution policy, namely *last-writer-wins*. The value of an update in repository memory will be indiscriminately clobbered by any later value that is checked in, even if there was no intervening

[2] Note that if the **`checkin`** on line 11 occurred before the **`checkout`** on line 00, register **`R0`** would always observe the value one.

checkout. In contrast, systems like CVS provide complex merging policies that may even rely on user intervention. In ASM-CMP, conflicts are resolved at the byte granularity.

We chose the last-writer-wins policy for several reasons. First, it is the same policy provided by coherent shared memory that programmers are already familiar with. When two updates race in a coherent system, the earlier version will always be clobbered. Software in coherent systems deal with this property by synchronizing updates to shared locations, and the same holds true in acoherent systems. Second, choosing to support only a last-writer-wins policy may simplify ASM implementations, as a last-writer-wins policy does not require any conflict detection mechanism or callbacks to threads. Finally, we will show that under most circumstances (notably for data race free software), conflicts do not occur, and so providing complicated resolution policies would seem to be a case of over engineering.

The two subtleties above hint at a common misunderstanding about the semantics of acoherence. The **`checkout`** and **`checkin`** operations are not synchronization operations, and execution between **`checkout`** and **`checkin`** cannot be considered a transaction. In this respect, checkout and checkin are more similar to memory fence operations in weakly consistent systems [2,160]. Because checkout and checkin cannot be used by themselves to achieve exclusion, programs using ASM need to use conventional synchronization mechanisms like locks and barriers in coordination with checkout and checkin. Details of how synchronization abstractions and checkout/checkin might interact depend on how software uses ASM. A few possibilities are discussed in the case studies later in this chapter.

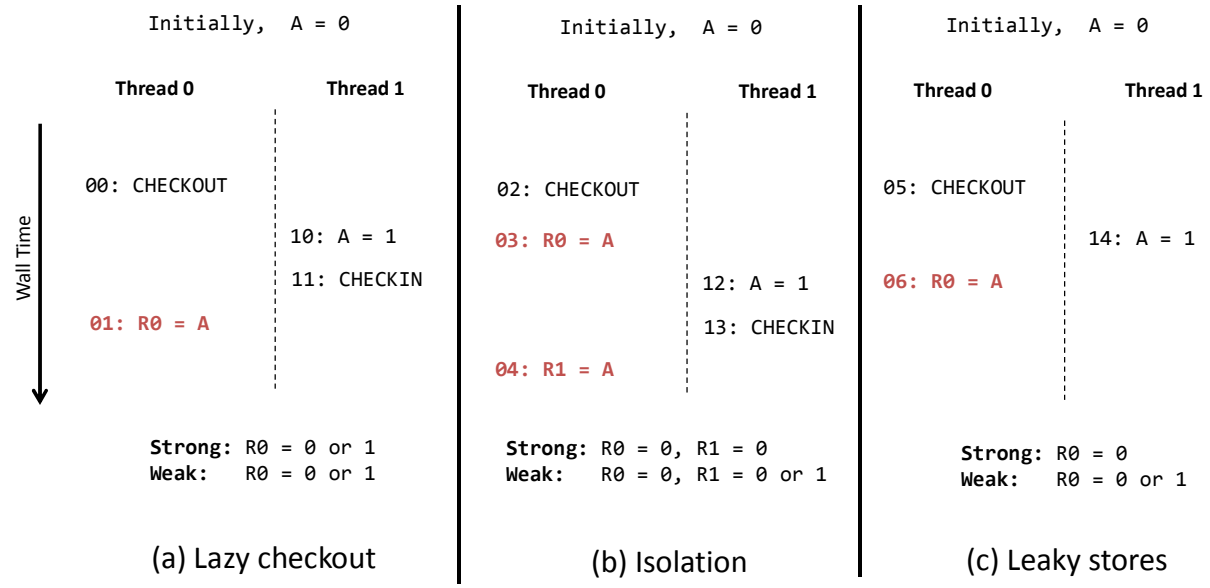

Figure 2-3. Semantics of acoherence.

(a) Lazy checkouts apply to all acoherence variations. (b) Strong and best-effort acoherence guarantee local isolation after the first reference following a checkout. (c) Weak acoherence provides no isolation guarantees.

2.1.2 Managing Finite Resources

In the descriptions above we have been purposefully vague on the details of working memory. In this subsection we discuss three variations of acoherence that each use a different model of working memory, namely *strong acoherence*, *best-effort acoherence*, and *weak acoherence*. Each variation presents slightly different semantics to software, though, as we will show, all are useful in different ways.

An ASM implementation can choose to implement all or a subset of the variations. Our initial prototype discussed in Chapter 4, for example, only provides best-effort and weak acoherence. If an implementation were only to support one type of acoherence, it would most likely be weak acoherence, for the reasons given below.

Strong Acoherence In the strong acoherence model, software can assume that working memory is the same size as repository memory, i.e. the entire address space. This means that in strong acoherence, working memory can hold any number of writes performed between a

`checkout` and the next `checkin` by a single thread. A strong acoherent segment guarantees that a local update will be hidden from remote threads until a checkin operation is performed and thus provides a form of isolation. The semantics of lazy checkout (Figure 2-2) still apply, though, and so this form of isolation is notably different from isolation in the (perhaps) more familiar isolation of database transactions [75,76]. In particular, using the terminology of Gray and Reuter, strong acoherence allows lost updates but does not allow either dirty reads or unrepeatable reads. As a result, strong acoherence is not equivalent to any of the four degrees of database isolation, and does not provide serializability.

We show an example of isolation in strong acoherence in Figure 2-3b. Thread 0 attains isolation on the location `A` by performing an access (line 03) following a `checkout` (line 02). After that point, actions by remote threads (e.g., lines 12-13) cannot affect the local version of location `A`, so that the value of `R1` will always be zero. Note that the isolation is only acquired after the first access, which is why the value of `R0` in Figure 2-3a is nondeterministic.

The isolation guarantee provided by strong acoherence can be used by software in a variety of ways. For example, software can use strong acoherence to achieve fast and efficient speculation. Because working memory in strong acoherence has perfect buffering, software can use working memory as a speculation buffer, completely eliminating the issue of version management. In contrast, software speculation in conventional cache coherence typically requires explicit data copies into temporary storage [82].

Strong acoherence can also be useful to programmers when debugging code. In strong coherence, races cannot occur between any local update and the next checkout/checkin. Thus, programmers can reason about bugs locally without having to consider the possibility that a location changes spontaneously because of an update from another thread.

Along the same lines of reasoning, strong acoherence can also reduce nondeterminism in parallel applications by reducing the number of possible races. Strong acoherence does not completely eliminate nondeteminism, though, because checkout/checkin can still race and because races can still arise between the first load after a checkout and a remote checkin (e.g., in Figure 2-3b).

While strong acoherence is a nice abstraction to reason about, the potentially large amount of buffering that is required may prove prohibitive for implementations. In the worst case, an implementation would need to buffer *number of threads + 1* (working memories plus repository memory) copies of the address space. It may be possible to emulate strong acoherence on top of best-effort acoherence, which, as we discuss next, has limited buffering requirements.

Best-effort Acoherence While we have shown strong acoherence is likely impractical to implement because of the potential buffering requirements of an application, in many cases the actual buffering requirements of a particular application may be manageable. Best-effort acoherence is an all-or-nothing model that exploits this observation. When the actual buffering requirements fall below an implementation defined maximum, best-effort acoherence is identical to strong acoherence. When the buffering requirements exceed that maximum, the model fails gracefully by performing a spontaneous checkout (i.e., undoing all local updates in working memory) and notifying software via an exception. This mechanism is similar to those proposed for hybrid transactional memory systems [18,50,106,130,153].

Best-effort acoherence can be used to emulate strong acoherence, and thus has all the same benefits of strong acoherence listed above. The catch, of course, is that the emulation may come with a significant performance penalty if the required amount of buffering is large. To emulate strong acoherence, software (likely a low-level runtime) must be able to emulate isolation in

working memory. One simple way to accomplish this would be to serialize execution by grabbing a global lock after failure.[3] We provide more details on the design of one such runtime, a simple software transactional memory manager, in Section 2.4.

Weak Acoherence Weak acoherence deals with limited buffering capability by allowing values in working memory to "leak" into repository memory. In the weak acoherence model, one can think of working memory as a limited capacity cache of repository memory.

We show the difference between weak acoherence and best-effort/strong acoherence in Figure 2-3b and Figure 2-3c. Unlike best-effort/strong acoherence, weak acoherence does not provide any isolation guarantee. When accessing a weak acoherent location, a thread may see the value change spontaneously between two reads (e.g., lines 03 and 04) if there is a race with another thread. Also, stores are not guaranteed to be hidden from remote threads until a checkin under weak acoherence. Thus, locations can race even if the remote data has not been checked in, as shown in Figure 2-3c.

While the leakiness, and hence uncertainty, of weak acoherence may seem like a considerable conceptual hurdle for rational software designers to overcome, we point to two anecdotes for why it is not. First, consider that *a valid implementation of weak acoherence is conventional cache coherence*.[4] In other words, because the weak acoherence model allows values to leak at any time, it would be valid to bypass working memory altogether and write/read directly into repository memory. Thus, even though the uncertainty that arises from leaks in working memory may seem complex to reason about, it is the exact same situation that shared memory software has already been dealing with for decades. In fact, those familiar with memory consistency

[3] Serializing execution actually leads to an execution model that is strictly stronger than strong acoherence. A more intelligent emulation may be able to avoid serialization by, for example, using a copy-on-write scheme.

[4] Of course, we believe hardware would be remiss to ignore the semantic benefits that acoherence provides by implementing full conventional cache coherence.

models may notice that in weak acoherence, **`checkout/checkin`** are similar (though not identical, see Chapter 3), to **`acquire/release`** in release consistency [70].

Somewhat paradoxically, while weak acoherence is comparable to conventional cache coherence it is also equivalent to strong acoherence under "normal" circumstances. If a program using weak acoherence is data race free, we will prove in Chapter 3 that the programming model is indistinguishable from strong acoherence. The intuition is that if data in a weak acoherent address space does not race (i.e., accesses to the same location by different threads are separated by checkout/checkin), then the leaks to repository memory will never be observed. Consequently, if weak acoherence is used with synchronized data, programmers can reason about it with the strong acoherent model in mind.

Because of its relaxed requirements, weak acoherence will likely be the easiest of the three variations to implement. It maintains the benefits of acoherence in general, notably the explicit communication of semantic information, and is well suited to data race free execution. In Section 2.4 we will show ways that software can use weak acoherence.

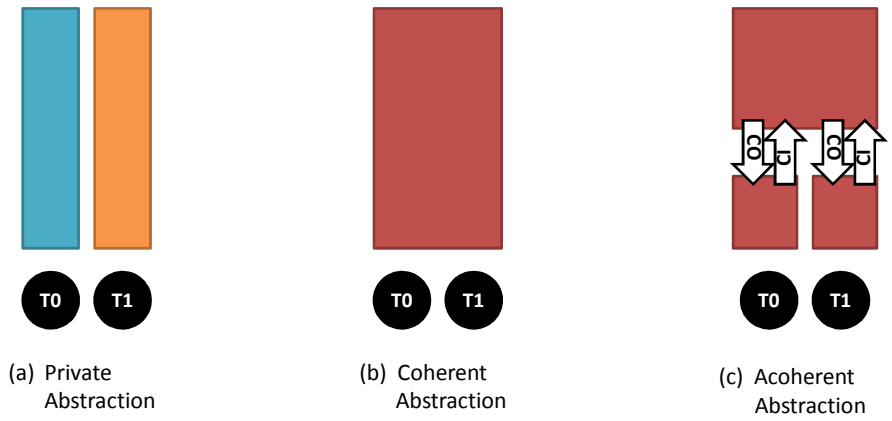

Figure 2-4. Memory abstractions available in ASM.
Different colors represent different address spaces.

2.2 Semantic Segmentation

While the acoherent model addresses the conceptual mismatch between synchronized data and the coherent shared memory abstraction, it leaves two issues unresolved. First, acoherence does not help software communicate the diversity of semantics found among data within a single program. Second, acoherence as described thus far, specifically in relation to the whole address space granularity of acoherent operations, is likely too blunt of an instrument to be useful. ASM solves both of these issues with semantic segmentation.

Segments collectively represent a mutually exclusive and exhaustive partitioning of an application virtual address space, such that every location in memory belongs one and only one segment. Segments do not have to consist of contiguous addresses, though an implementation may choose to enforce such a restriction[5]. Software controls segment creation, and can partition the address space in whatever manner it deems convenient. Because segments are virtualized, an ASM implementation will likely protect segment creation with a system call (see Chapter 4).

[5] Our initial prototype, ASM-CMP (Chapter 5), forbids noncontiguous segments. Other implementations could ease that restriction, thereby giving more flexibility to software.

Table 2-1. Segment types and qualifiers

Type	Available Qualifiers
Private	-
Coherent	Read-Only
	Read-Write
Acoherent	Strong
	Best-Effort
	Weak

In ASM, every segment has an associated semantic abstraction that defines the behavior of reads and writes to the segment. A program can have as many segments as it desires, and every one of those must be independently assigned an abstraction. Currently, we define three different semantic abstractions, shown in Figure 2-4. Additionally, two of the abstractions can be associated with qualifiers that further refine the semantics of data in the segment. Table 2-1 lists all abstractions and their qualifiers, if any.

The first abstraction, *private* (Figure 2-4a), presents memory as a shared nothing resource to each thread. This abstraction is similar to model found in a distributed system with private memories. Private segments can be used, for example, to hold stack data or dynamically allocated data that is used for temporary local storage. Each private segment can only be accessed by one thread, and attempts to access the segment from another thread in the same process will result in an exception.

The *coherent* abstraction (Figure 2-4b) is the same one present on most modern multicore systems. Software sees memory as a single, flat black box in which all updates become visible immediately to other observers. In ASM, accesses to coherent memory are also guaranteed to be sequentially consistent with respect to one another (see Chapter 3). Coherent segments are qualified as either read-only or read-write. The read-only qualifier is an indication from software

that no write operations will occur in that segment. The prototypical read-only segment is the text (i.e., instruction) segment of a program.

The read-write coherent abstraction can be useful for data in which it is critical that updates become visible immediately, notably low-level synchronization variables. As we've shown in Chapter 1, the types of variables that are a semantic match for coherence are typically a small fraction of variables in an application, and so we expect the read-write coherent abstraction to be used sparingly.

Finally, ASM gives software the option of viewing memory in an acoherent organization (Figure 2-4c) that was introduced in the previous subsection. Acoherent segments are qualified with the acoherence variation (strong, best-effort, or weak) that software wants to use.

When a thread checks out or in, it always does so at the segment granularity. In other words, in ASM the repository memory address space is defined by a segment. Using segments as the granularity for acoherence operations achieves a balance between simplicity and flexibility. On the one hand, it keeps things simple for software by not forcing it to **`checkout/checkin`** individual memory addresses. On the other hand, it still gives software some ability to selectively apply acoherence operations.

Segment types and qualifiers can change over the course of an execution. A runtime can, for example, load a program's text into a coherent read-write segment and then convert it to read-only before entering the application. Segment properties can only change when the segment is in a quiesced state. Notably, for an acoherent segment this means that no thread has the segment checked out. Property changes may result in data movements in the underlying physical memory storage. For example, when transitioning from a private segment to a weak acoherent segment, the private segment may be flushed to globally visible memory so that the most recent updates

can be seen by all threads. Property changes will likely be long-latency operations in an implementation and should be used sparingly (e.g., at the beginning or end of an execution phase).

2.3 System Considerations

The ASM model assumes that a system is capable of dividing memory into segments. The word "segments" evokes strong reactions in many people who may be familiar with programming on segmented architectures with multidimensional memory (e.g., [45,94]). While traditional memory segments would be a valid way to implement ASM segments, it is not a requirement. In a system using paging, for example, one could use a variety of techniques to break virtual memory regions into segments. One option would be to use a mechanism similar to the Memory Type Range Registers (MTRRs) in x86 that apply characteristics to memory regions [93]. MTRRs are currently used, for example, to mark regions as unchachable or write-combining, and extending them to support ASM types would be relatively straightforward. One could also achieve more fine-grained segments than are typically used by MTRR regions using a fine-grained memory protection mechanism like Mondrian [178]. Others approaches not use in existing systems are also possible. For example, the ASM-CMP prototype in Chapter 4 uses a novel segment architecture that uses traditional memory segments but maintains a flat, one-dimensional address space.

System software will likely want to virtualize segment creation and modification. That way system software can control how virtual memory is laid out in physical memory and could even let processes share data by mapping two virtual segments to the same physical segment. System

software will also want to control segment modifications to avoid any consistency issues that might arise from multiple threads changing segment attributes at the same time.

The semantics of checkout and checkin can complicate a context switch. This is especially true in strong acoherence in which the buffered private memory has to survive the context switch. There are several options for dealing with this constraint. First, if a system trusts the application, it can coordinate checkout/checkin so that there are no checked-out segments at context switch time. Such a scenario may happen in the embedded space or even in large scale datacenter-type applications where a single application runs on a machine that is vertically managed by a single entity [124]. In the absence of trust or with an application where it would not be easy to quiesce checkins, a strong acoherent implementation would need to save and restore private memory state on a context switch. To reduce the overhead of saving private memory state, the implementation could repurpose any use tracking mechanisms that may already be present to perform **`checkout/checkin`** operations (e.g., those in Chapter 4). Such mechanisms could allow the system to avoid saving state that could be easily reconstructed after a context switch (e.g., state that hasn't been written yet).

Luckily, context switches for best-effort and weak acoherent segments are simpler to perform. There are two options for dealing with a checked-out best-effort segment during a switch. The system can either save the segment state using one of the policies described above for strong acoherence or it can simply choose to let the best-effort mechanism kick in and checkout the segment. The system can chose dynamically which mechanism to evoke depending on the characteristics of the application, state of the segment, etc. If the system chooses to kill the segment, it could delay the exception notifications until the process is rescheduled.

Weak acoherent segments are the easiest to deal with on context switches. If the process is pinned to a particular physical private memory (e.g., an L1 cache in ASM-CMP, Chapter 4), then the system does not need to take any action. After the context switch, data from the weak acoherent segment may spill into global memory but that behavior is expected by the application anyway. If the process is not pinned and may resume execution using a different physical private storage, then the system needs to flush updated data to global memory by invoking a checkin operation. The system does not have to wait for the checkin to complete before completing the context switch. Rather, it needs to ensure the checkin completes before the process is rescheduled elsewhere in the system.

Application software may find it useful if some of the underlying **`checkout/checkin`** mechanisms are exposed. For example, a runtime implementing software transactional memory on ASM would benefit from access to the read and write sets that an implementation may be tracking for a segment. If an implementation chooses to expose those mechanisms, the system would be responsible for managing accesses to those resources. In ASM-CMP we advocate a mechanism similar to SoftSig [169] that gives user software access to read and write set signatures. Other mechanisms may be possible as well, including those found in hardware isolation systems [86,169].

2.4 Using Acoherence

In this subsection we show how software might use the acoherence abstraction by providing examples for three common data sharing patterns. After establishing this generic base, we will expand the usage examples to ASM as a whole in the next subsection when we discuss four different case studies that highlight how complete applications might be built.

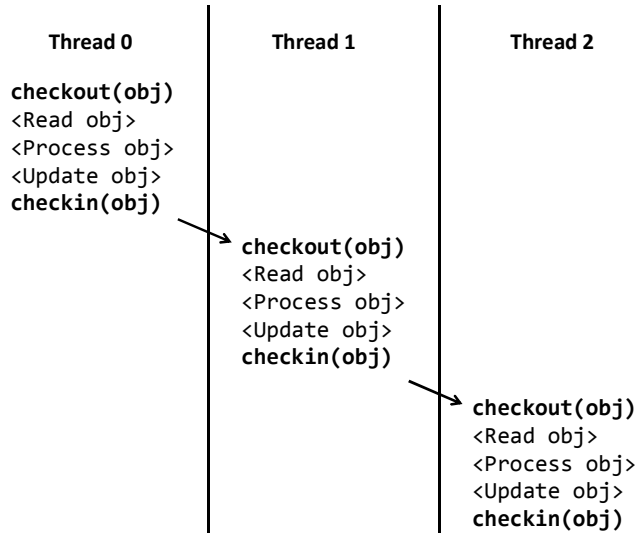

Figure 2-5. Example using acoherence with a migratory object.

The examples in this subsection highlight the versatility of acoherence. Using the same basic acoherent model, software can communicate at least three different (and common) access patterns to the underlying hardware. Hardware can then use that information to ensure that it is managing data in the most efficient way possible in a given implementation.

2.4.1 Migratory Sharing

The migratory sharing pattern applies to data that is manipulated by only one thread at a time [79]. Objects undergoing migratory sharing are read by a thread, processed, and then subsequently updated before being passed to the next reader. Data protected in a critical section most likely adheres to the migratory pattern.

We show a migratory object in Figure 2-5. For simplicity, assume the object fits entirely in the segment and that no other object is held by the segment. After acquiring the object, a thread checks the object out to ensure that it will see the latest updates. When the thread is done performing its updates on the object, it checks it back in before passing it on to the next thread. Assuming the object is correctly synchronized, the example in Figure 2-5 will behave identically under either strong or weak acoherence because there are no data races on the object's members.

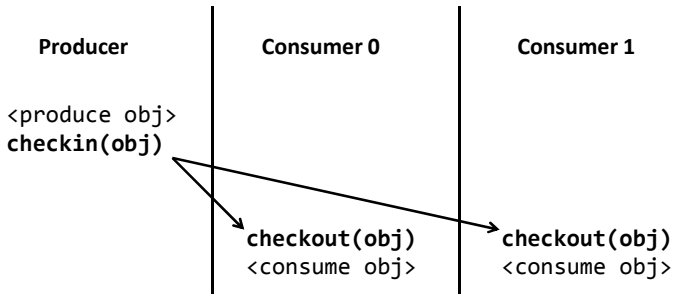

Figure 2-6. Example using acoherence with producer/consumer sharing.

Figure 2-5 is an example of symmetric acoherence. There is exactly one checkin for every checkout and objects are never checked out or in twice in a row. Hardware, if it so chooses, could detect the symmetry of acoherent operations to determine that a segment holds migratory data.

2.4.2 Producer/Consumer Sharing

Data that adheres to the producer/consumer sharing pattern is written by one and only one thread and is read-only by one or more (different) threads. This pattern might arise, for example, in pipelined applications where each thread is responsible for a single step of a multi-step process.

We show how software might use acoherence with producer/consumer data in Figure 2-6. The producer thread generates a data object, checks it in, and then notifies consumers (e.g., with a condition variable) that the object exists. A consumer, upon learning of the object, checks it out before trying to access any of the object members. Like the migratory sharing example, if the producer and consumer threads are properly synchronized, this example behaves identically for either strong or weak acoherence.

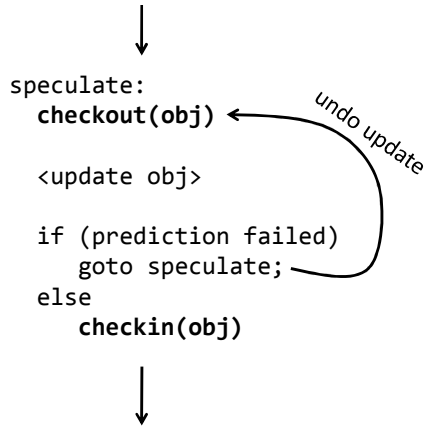

Figure 2-7. Example using acoherence with speculation.

Figure 2-6 is an example of asymmetric acoherence. There are an unequal number of **`checkout`** and **`checkin`** operations performed over the course of execution (in this example, there are twice as many **`checkout`**s as **`checkin`**s). The producer thread only checks the object in and consumer threads only check the object out. These consecutive acoherence operations are safe because the data is write-only/read-only by the respective threads.

2.4.3 Speculative Sharing

Software, often as a low-level runtime layer, may want to speculate past a predictable yet long-latency decision in order to improve performance. When doing so, it must be able to undo speculative updates in case of a misprediction. In conventional cache coherent software, this most likely will require an explicit data copy to preserve the original value before applying a speculative update. In a system with acoherence, software can avoid such copies, potentially leading to a significant performance enhancement. This example requires the isolation properties of acoherence, and so it does not apply to weak acoherence.

We show an example of a software speculation routine in Figure 2-7. Assume that the code is identical on all threads that try to speculate. When speculation begins, a thread checks out the

segment holding the object. This `checkout` operation ensures that the thread will see the most up to date version of the object. After updating the object, the thread checks if the speculation prediction was correct. If it was, a `checkin` operation suffices to make the update non-speculative by publishing the update to all threads. Otherwise, the thread performs a consecutive `checkout` operation, which, because it creates a copy of repository memory, effectively undos the update in one fell swoop.

If the segment holding the object is a best-effort acoherent segment, then software would also need to implement an exception handler in case the best-effort hardware exhausts buffering resources. The handler might redirect the thread to a `dont_speculate` function that that checks the prediction up front, making a spill from working memory acceptable. While executing `dont_speculate`, software would also need to suppress best-effort exceptions.

2.5 Case Studies

In this subsection we discuss four case studies that highlight different ways that programmers can use the ASM model. First, we show how an existing pthread program can work in ASM without any application-level changes. We'll also show how this technique can be extended to all pthreads programs in general as long as two simple guidelines are followed. The pthreads case study uses ASM as a blunt hammer by using coarse-grained segments and overly conservative segment management in order to achieve application compatibility. In the second and third case studies we show how a programmer might use ASM more surgically by using many fine-grained segments and intelligent segment management. The second case study shows how a programmer might use ASM for statically partitioned data sets and the third explores the more interesting case of dynamically partitioned data. The final case study shows how ASM might be used as a

```
01: pthread_barrier_t barrier;              // allocated in global segment
02: char* shared_data;
03:
04: int main(int argc, char* argv[]) {
05:   int i,j,k;                            // allocated in stack segment
06:   pthread_t sib[N];
07:   shared_data = malloc(PROBLEM_SIZE);   // allocated in heap segment
08:   pthread_barrier_init(&barrier, NULL, 2);
09:
10:   for(i=1; i<N; i++)
11:     pthread_create(&sib[i], NULL,       // check in/out heap,global
12:                    worker, (void*) i);
13:   worker((void*) 0);                    // main thread becomes worker
14:   for(=1; i<N; i++)
15:     pthread_join(sib[i], NULL);
16:   return 0;
17: }
18:
19: void* worker(void* arg)
20: {
21:    while (work remains) {
22:      <split work>
23:      <do work>
24:      pthread_barrier_wait(&barrier);   // check in/out heap, global
25:    }
26: }
```

Figure 2-8 – Example pthreads program using ASM.

substrate to achieve efficient software speculation by detailing the design of a simple software transactional memory runtime. This final case study is the only one to take advantage of best-effort/strong acoherent segments.

2.5.1 Case Study #1: Existing *pthreads* Program

In this case study we show that many existing applications designed for coherent shared memory are already written in an ASM-compatible style, and as such can *work in an ASM system without application source code changes*. We start with the existing application shown in Figure 2-8 that represents a simple data parallel application written using the pthread library [92] (e.g., like **`fft`** [180]). After initialization, the program creates **`N-1`** worker threads. The master thread itself then acts as worker thread, for a total of **`N`** workers. The execution proceeds in a series of global steps separated by a barrier. At each step, a worker acquires a portion of the data

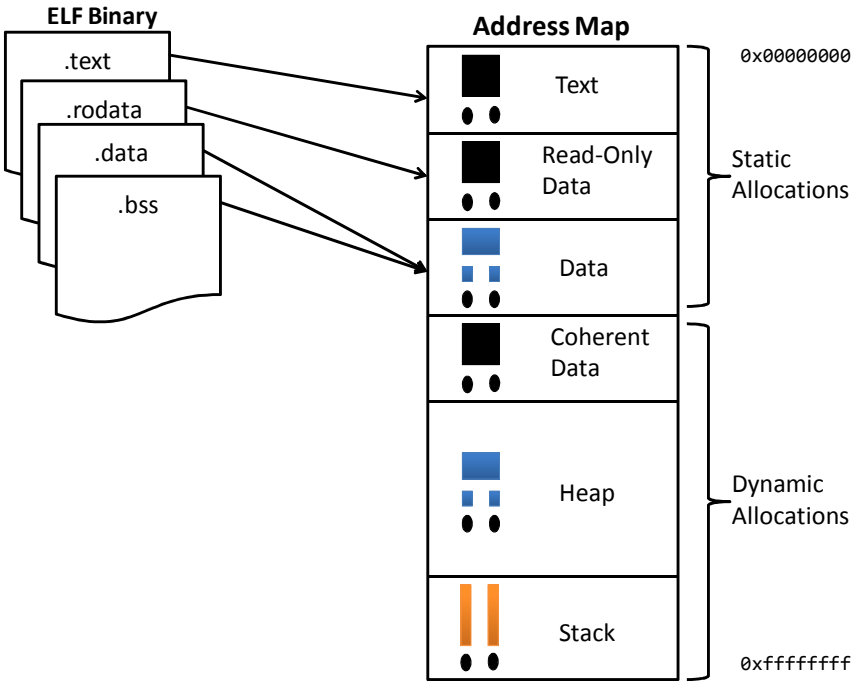

Figure 2-9. Example C/pthread segment mapping.

set, performs the necessary operations, and then writes any updates to the shared work space. We assume that the program is data race free.

To convert this application so that it runs on an ASM system without modification, we exploit two observations regarding pthread programs. First, pthread programs are already segmented by the C runtime into regions such as the heap and stack, which we can reuse to create ASM segments. Second, pthread programs already explicitly identify communication points using synchronization, which can be reused to call **`checkout`**/**`checkin`** instructions. In our example, these points occur at the barrier inside the worker function (line 24). In the remainder of this case study we will explore both of these exploitations in more detail.

Step 1: Segment Management We automate segment management by observing that pthread programs, as an artifact of being C programs, are already segmented. All C programs break data into segments that hold text, read-only data, global data, the stack, and the heap (i.e., static segments such as **`.text`**, **`.rodata`**, **`.bss`**, **`.data`**, etc. in ELF binaries [166] and dynamic

```
int pthread_barrier_init(pthread_barrier_t* barrier,
                         const pthread_barrier_attr_t* attr,
                         unsigned count)
{
   …
   barrier->_barrier = coherent_malloc(unsigned);
   …
}
```

Figure 2-10. ASM pthread barrier initialization.

segments like the heap and stack). A program loader can easily map these program segments into ASM segments at runtime. Text and read-only data go into a single read-only-coherent segment, global data and the heap each go into a weak acoherent segment, respectively, and each thread's stack goes into its own private segment. We only show one stack in Figure 2-9 for simplicity, though in reality one exists for each thread in the program.

In addition to the standard segments found in conventional coherent pthread programs, the ASM implementation of pthreads uses one additional segment. This segment, which is a coherent read-write type, is used to hold low-level synchronization variables that prefer the update-immediate semantics of coherence. In our example, the word(s) representing the barrier state are allocated in the coherent segment. If the implementation is a basic counting barrier, the coherent state would just include the single counter. More complicated barrier implementations, such as a tree [128], may include more shared state. In either case the allocations occur inside the `pthread_barrier_init` function, as shown in Figure 2-10 for a counting barrier.

Step 2: Checkout/Checkin Like segment management, `checkout/checkin` placement can be automated in pthread programs. Consider the example program, in which threads don't intend to communicate any updates until they reach the timestep barrier; threads effectively publish their updates by signaling their arrival to the barrier. This communication pattern is well structured and is explicitly identified with barrier wait calls.

```
int pthread_barrier_wait(pthread_barrier_t* barrier)
{
   …
   checkin(heap_segment);
   checkin(global_segment);
   <wait for barrier>
   checkout(heap_segment);
   checkout(global_segment);
   …
}
```

Figure 2-11. ASM pthread barrier wait implemention.

We can exploit the communication intentions revealed by synchronization calls to automatically insert **`checkout`**/**`checkin`** instructions. In the example, the correct place for **`checkout`** is immediately before exiting the barrier wait function before the next timestep begins, and the correct place for **`checkin`** is immediately after entering the barrier when the current timestep ends. Because the library lacks information about what data is actually being communicated (just that it is being communicated), we apply **`checkout`**/**`checkin`** to both the data and heap acoherent segments. The resulting **`pthread_barrier_wait`** implementation is shown in Figure 2-11.

While our example just covers a barrier, the same principle applies to all pthread synchronization routines (e.g., **`checkout`** after acquiring a lock, **`checkin`** before unlocking a lock). Because checkout and checkin are being automatically inserted for these programs, there may be **`checkin`**s that are not associated with any updates. This could happen, for example, when a thread checks in the data segment even though it is only modifying the heap. We discuss the implications of these useless operations on an implementation in Chapter 4.

There are a few additional checkout/checkin instructions placed automatically in the example program. Inside of **`pthread_create`** the master thread checks in all of its acoherent segments before performing the clone operation so that children will see all previous updates. Also, there

is an implicit `checkout` at the beginning of execution for every thread and an implicit `checkin` when a thread completes.

While many pthread programs will work as is like the example in this case study, there are some special circumstances that will require small application-level changes. One such circumstance is an unsynchronized shared flag. As others have previously observed, however, the use of such flags in pthread programs is rare, perhaps due to the fact that flag synchronization is not portable across systems with different memory consistency models [120,143]. To use flags in ASM, they should either be paired with a synchronized condition variable (preferred) or allocated in a coherent segment.

Another situation that requires application changes are programs that share data on the stack, which in the transformation methodology described above, can only hold private data. In our experience this situation occurs most often when a variable created on a master thread's stack is shared with children. The fix is simple, and just requires moving the shared allocation to the heap.

Extensions There are several ways that one could build on the basic transformation described above to make better use of the features of the ASM programming model. First, a program could distinguish between private and shared heap allocations. Doing so could relay better semantic information to hardware, which presumably could then do a better job managing data. A program making the distinction between private and shared dynamic allocations might use a special `private_malloc` function that allocates from a private segment.

A program could also use segments more specifically than is done with the automatic transformation. This could avoid the useless operations described above and could reduce any

```
01: double **matrix;
02:
03: int main(int argc, char* argv[])
04: {
05:   grid2d = new_segment(NROWS*sizeof(double*), RW_COHERENT);
06:   for(i=0;i<NROWS;i++)
07:     matrix[i] = new_segment(NCOLS*sizeof(double), WEAK_ACOHERENT);
08:   make_read_only(matrix);
09:   <populate initial values>
10:
11:   spawn_workers(NPROCS);
12:   wait_for_workers();
13: }
14:
15: void worker(void* args)
16: {
17:   do {
18:     for (row = row_start; row != row_end; row++) {
19:       if (row == row_start || row == row_end)
20:         checkout(matrix[row]);
21:       for (col=0;col<NCOLS;col+=2) {
22:         matrix[row][col] = f(grid2d[row][col-1], grid2d[row-1][col],
23:                              grid2d[row][col+1], grid2d[row+1][col]);
24:       }
25:       if (row == row_start || row == row_end)
26:         checkin(matrix[row]);
27:     }
28:     barrier_wait()
29:   } while (convergance() == false);
30: }
```

Figure 2-12. Pseudocode for a 2D stencil algorithm written for the ASM model.

contention bottlenecks that may exist in an implementation between checkout/checkins to the same segment.

For performance and scalability reasons it may be beneficial to create more than one heap segment in order to have a more fine-grained checkout and checkin. In our experience so far one heap has been sufficient, but if needed compiler techniques like thread escape analysis can help automate management of a multi-segment heap [154].

2.5.2 Case Study #2: Data Parallel with Fine-Grained Segment Control

In this case study we explore a program that is written from the ground up for the ASM model and compare it to the pthread program from the previous case study. The target algorithm is a

generic 2D stencil code that might, for example, be found in the kernel of a linear system solver [68]. The algorithm iteratively calculates new values of a matrix location based on the values of four (north, east, south, west) neighbors and itself until convergence is reached. The matrix is colored in a checkerboard manner, and iterations alternate between updating black and reading white and updating white and reading black so that inter-iteration updates do not affect each other. The work is split among threads by assigning a contiguous group of matrix rows to each thread. Using that split, only the data in the edge rows of a group are communicated between processors.

Figure 2-12 shows C-inspired pseudocode for the ASM version of the algorithm. As we did above for the pthread case study, we will break down the algorithm generation into two steps, namely segment management and checkout/checkin.

Step 1: Segment Management Like the pthread example above, some segment assignment is handled automatically by the runtime. The program text is automatically assigned to a coherent read-only segment and the global data is put in a weak acoherent segment before the program starts. Each thread is also automatically assigned a private stack. In a notable difference from the first case study, the native ASM runtime does not automatically create a global heap. Instead, the application guides allocations by creating new segments with varying types for the dynamic shared data it requires.

The stencil algorithm contains one main data structure, namely the shared matrix. The pointer to the matrix is allocated by the runtime into the data segment. Because the pointer is never updated after initialization, though, worker threads do not need to worry about managing communication in the data segment as it is effectively read-only. The matrix itself is composed of many different segments. The first dimension of the matrix, which is a single segment, holds

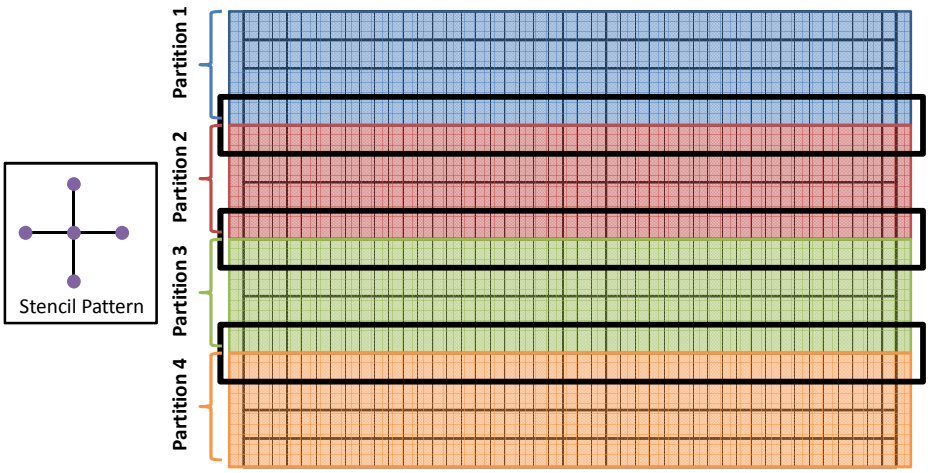

Figure 2-13. 2D stencil partitioning.

Each row of the grid is allocated into its own segment. Rows are partitioned among threads by splitting the matrix into equal horizontal cuts. Solid box outlines show the rows that are communicated between timesteps.

pointers to each of the rows and is allocated in a coherent read-only segment because the pointer values never change after initialization. The segment is actually created as a coherent read-write (line 05) so that initial values can be set, and is then updated later to read-only (line 08). Each row of the matrix is allocated as a weak acoherent segment. We chose this type because the data in rows are shared, will change over time, and are properly synchronized. When the rows are split up among threads, many rows may actually be private to a thread over the course of the execution and could be allocated in a private segment. For simplicity, however, we do not consider such an optimization here (though the stencil workloads such as **`heat`** used in the evaluation in Chapter 5 do).

Step 2: Checkout/Checkin We chose to split the rows into separate segments so that the implementation could assert more fine grained control over checkout and checkin operations. Because only rows on the edges of the matrix partitions communicate, they are the only rows that need **`checkout/checkin`** management. At the beginning of each iteration, edge rows are

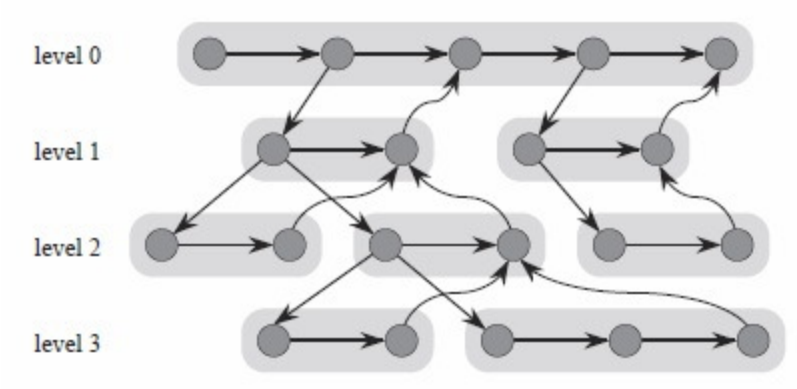

Figure 2-14. The Cilk execution model.

Downward edges correspond to child tasks and horizontal edges within a shaded group correspond to continuations of the same parent task. Image credit: Blumofe, et al. [26].

checked out so that updates from the previous iteration are seen. At the end of each interation the edge rows are checked in to publish the new updates.

By using fine-grained segment management (at least compared to the conventional pthread conversion approach), the ASM software communicates more semantic information to the underlying hardware. For example, it communicates what subset of the data structure is actually shared after an iteration. An implementation could use this information to optimize data movement, e.g., by skipping global updates for data that remote threads will never read.

2.5.3 Case Study #3: Task Parallel with Fine-Grained Segment Control

In this case study we again visit the idea of using fine grained segments with ASM, but this time apply it to a task parallel application with irregular sharing. The core of the case study is the design of a parallel work stealing task queue. In particular, we will design scheduling runtime modeled after Cilk [26] and then use it to create a simple program that (inefficiently!) calculates the n^{th} Fibonacci number.

This case study highlights several ASM programming techniques not present in either of the two previous studies. First, the task queue design uses asymmetric segment management, in

which the number of **`checkouts`** is not equal to the number of **`checkins`**. This arises because the producer/consumer nature of the task queue is translated intelligently with ASM operations. This is especially in contrast to the pthread case study that effectively breaks a program into read-modify-write execution chunks no matter what the software is actually doing. Second, in this case study we explore the idea of split data structures, in which different elements from the same data structure are allocated in different segments. Finally, we will also show how ASM can be used to implement the same abstract data structure in different ways to match actual usage patterns. In particular, we will show two ways that a shared queue can be implemented that have different contention characteristics.

2.5.3.1 ***Background on Cilk***

Before diving in to the details of the runtime design, we first provide a brief introduction to Cilk. Knowledgeable readers may skip ahead to Section 2.5.3.2. Cilk is the combination of a language extension for C (and in later versions C++ [65]) and a provably efficient runtime scheduler. Programs written for Cilk express a form of fork-join parallelism, such that an execution can be represented as a directed acyclic graph of tasks as shown in Figure 2-14. A parent task can be dependent on the return value of a child task, but all other tasks in the graph are assumed to be mutually independent. This independence is the source of parallelism in Cilk programs. Tasks are created using a special **`spawn`** keyword that transforms a normal function call into a child task of the caller. The child is free to execute immediately (downward edge in Figure 2-14) and the parent is free to continue execution without waiting (horizontal edge in Figure 2-14) for the child to complete. If the parent needs a value produced by the child, it can use the **`sync`** keyword (corresponding to an upward edge in Figure 2-14) that suspends the parent task until all of its children have completed.

The **spawn** and **sync** keywords are transformed by the Cilk compiler into runtime calls. The runtime uses a work stealing queue to schedule tasks. In Cilk, each thread has its own queue used for pushing newly created tasks (created via **spawn**) and popping more work when the current task completes or suspends (via **sync**). If a worker's local queue is empty when it looks for more work it will attempt to steal a task from a remote queue. When stealing, tasks are taken from the *back* of the remote queue, which corresponds to the oldest task. The Cilk designers have shown that older tasks are more likely to generate more work, and thus reduce the probability of more stealing in the future. Blumofe et al. have shown that the Cilk runtime performs within a constant factor of optimal [26].

The success of the Cilk project has spawned (no pun intended) a variety of subsequent work that adds syntactic sugar and/or creates task runtimes for new platforms [36,65,78,115,129,151]. Of particular relevance to this case study, Min et al. [129] created a Cilk-like environment for a large distributed platform built in the UPC language [33]. Like the ASM environment, the UPC programming model has a notion of different types of memory (e.g., private and shared), and is a fact reflected in their task queue design. We borrow several ideas from that work in the design discussed below.

```
void taskq_init(int num_workers);
void taskq_finish();
void taskq_spawn(void* (*task)(void*), void* in, int in_size);
void** taskq_sync();
```

Figure 2-15. Task parallel library API.

*2.5.3.2 **Task Parallel Environment***

Unlike Cilk, in this case study we do not assume any compiler support for our application. Thus, instead of using primitive **spawn** and **sync** keywords to describe task relationships, our target application will make calls to the runtime directly using the library API shown in Figure

2-15. In addition to the syntactic niceties lost by calling the library directly, there are several subtle differences between these functions and their analog in Cilk. For example, the **taskq_spawn** function puts the child task on the queue and continues execution of the parent. In Cilk, the opposite would occur. None of the subtle differences impact the goal of our case study, though, and so won't be discussed further here. We do discuss them further in Section 5.3.2.4 when describing the workloads created from this API for the evaluation of our ASM prototype.

The **taskq_init** function needs to be called before any tasks are created. It creates **num_workers** worker threads and initializes the global runtime state, including the work stealing task queue. We consider the main thread (the one that calls **taskq_init**) a worker in its own right, and so **num_workers** must be at least one. The **taskq_finish** should be called when the program completes, and is responsible for cleaning up worker state.

A program using the library creates new tasks with the **taskq_spawn** routine. The first argument is the function that the newly created task will execute. Task functions take a single opaque pointer argument and return a single opaque pointer value, which is similar to the signature of a pthread thread [92]. The task input argument is passed as the second parameter (**in**) of the **taskq_spawn** routine, and the size of that argument is passed as the third parameter (**in_size**). The task created by **taskq_spawn** must be ready to execute when the routine is invoked.

A parent task can retrieve the return value of children tasks with the **taskq_sync** routine. **taskq_sync** suspends the calling task until all children have completed and then returns their results in an array. Unlike task inputs, which can be arbitrarily sized objects, tasks can only return a simple pointer to their parent with this API. **taskq_sync** also communicates the execution status of the progeny by hijacking an unused bit in the result pointer. This bit tells the

parent whether or not the child or any of the child's descendants was stolen. This feature can be particular handy for an ASM application that can use the information to decide if a **`checkout`** or **`checkin`** is required before continuing after the sync.

In Figure 2-16 we show how the Fibonacci number calculator described by the Cilk authors [26] is implemented using the task parallel API. To refresh, the nth Fibonacci number is the sum of the first n numbers in the Fibonacci sequence, i.e., $F_n = F_{n-1} + F_{n-2}; \quad F_0 = 0, F_1 = 1$. We calculate the number by turning each recursive calculation into its own task (function **`fib`** on line 01). When inputs to the task correspond to the base case, the task immediately returns its input (line 04). For inputs smaller than sixteen, we avoid creating a new task by recursively calling the fib function using normal recursion (line 06). This optimization, also done in [26], ensures that leaves in the execution's task dependency graph have enough work to yield a parallel speedup. Finally, the fib function will create additional tasks when the input is larger than sixteen (lines 08-16).

Because the only static task in the application returns an integer, this program does not have to perform any explicit checkout/checkin operations outside of those in the task runtime (discussed below). More complicated workloads, such as those described in Section 5.3.2.4, may require explicit **`checkout/checkin`**s, but for this case study we choose to focus on the task queue design itself.

```
01: void* fib(void* in) {
02:   int n = in;
03:   if (n < 2) {
04:     return n;
05:   } else if (n < 16) {
06:     return fib(n-1) + fib(n-2);
07:   } else {
08:     int n1 = n - 1;
09:     int n2 = n - 2;
10:
11:     taskq_spawn(fib, &n1, sizeof(int));
12:     taskq_spawn(fib, &n2, sizeof(int));
13:
14:     void** result;
15:     result = taskq_sync();
16:     return result[0] + result[1];
17:   }
18: }
19:
20: int main(int argc, char* argv[])
21: {
22:   int n, num_workers;
23:
24:   <read options into n, num_workers>
25:
26:   taskq_init(num_workers);
27:   taskq_spawn(fib, &n, sizeof(int));
28:
29:   void** result;
30:   result = taskq_sync();
31:
32:   printf("fib(%i) = %i\n", n, result[0]);
33:   return 0;
34: }
```

Figure 2-16. Fibonacci number calculator using the task parallel API. We omit typecasts required by C to keep the example succinct.

2.5.3.3 Work Stealing Task Queue Design

We have two goals for the design of the parallel work queue. First, we want to maintain the provable efficiency achieved by the Cilk runtime. Second, we want to be able to exploit the properties of ASM by using segments, **`checkout`**, and **`checkin`** in intelligent ways. To achieve these goals, we use the Cilk task queue as a reference design and then refine it for ASM.

Owner Worker/Victim	Remote Worker/Thief

```
01: push() {
02:   T++;
03: }
04:
05: pop() {
06:   T--;
07:   FENCE;
08:   if (H > T) {
09:     T++;
10:     lock(L)
11:     T--;
12:     if (H > T) {
13:       T++;
14:       unlock(L)
15:       return FAIL;
16:     }
17:     unlock(L);
18:   }
19:   return OK;
20: }
```

```
21: steal() {
22:   lock(L);
23:   H++;
24:   FENCE;
25:   if (H > T) {
26:     H--;
27:     unlock(L);
28:     return FAIL;
29:   }
30:   unlock(L);
31:   return OK;
32: }
```

Figure 2-17. Queue concurrency the THE protocol in cache coherent shared memory.

Assume the queue is implemented as an array and that H and T represent the head and tail positions, respectively. See Frigo, et al. for more details [67].

The Cilk runtime uses a distributed work queue. Each worker has its own queue, and to steal a worker must access the queue of a remote worker. As stated above, the owner of the queue always grabs the youngest task in the queue and thieves always steal the oldest task. In such a scheme true contention should be rare. First, the queues are accessed mostly by a single (the owner) thread, and are only accessed by thieves when they run out of work. Second, the owner and a thief operate on different ends of the queue, so even if their accesses are concurrent they may not be conflicting. Thus, the Cilk work queue uses an optimistic concurrency protocol based on a variant a Dijkstra's mutual exclusion algorithm [54] rather than heavyweight locks.

The resulting protocol, shown in Figure 2-17, is called the THE protocol [67]. In the common case of the owner worker accessing the queue without contention, no locks are acquired (lines 06-08). When contention does occur the protocol ensures that either the stealer or the owner will notice the contention and will take corrective action (lines 08-18 and 25-29). The protocol

serializes access to the queue by remote threads with a basic lock. When run on hardware that is not sequentially consistent, the THE protocol also requires two Store->Load memory fences (lines 07 and 24).

The ASM task queue modifies the basic THE protocol in two ways. First, it partitions each queue into a private portion and a shared portion. The private queue always holds the youngest tasks and the shared queue always holds the oldest tasks. The queues are logically connected such that the private queue head (oldest) and shared queue tail (youngest) are adjacent in chronological order. Workers can access their private portion of the queue without any concurrency protocol and remote threads can only steal tasks from the shared queue. By partitioning the queue, we avoid all concurrency overhead in the common case where only the owner is accessing the queue.

The potential downside of the design is that the partitioning may prevent a thief from stealing useful work if the owner thread is hiding all of its tasks in the private portion. We avoid this by setting a minimum occupancy on the shared queue. Periodically during execution, each worker checks the size of its shared queue and, if it finds less than the minimum number of tasks present, will transfer the oldest tasks in the private queue until the minimum is met. In practice we have found that a minimum occupancy of two is sufficient.

The second change that we make to the reference THE work queue is the addition of *localized* task pools. While we still use an array representation of the shared queue, the elements of that array are pointers to local task objects rather than global task frames as is done in Cilk [67]. Note that localized does not mean private; all workers can still access objects in all pools, but by convention there is one “owner”. A worker will only use task objects that are allocated out of its local pool, which, as we will show, simplifies segment management. The cost of localized

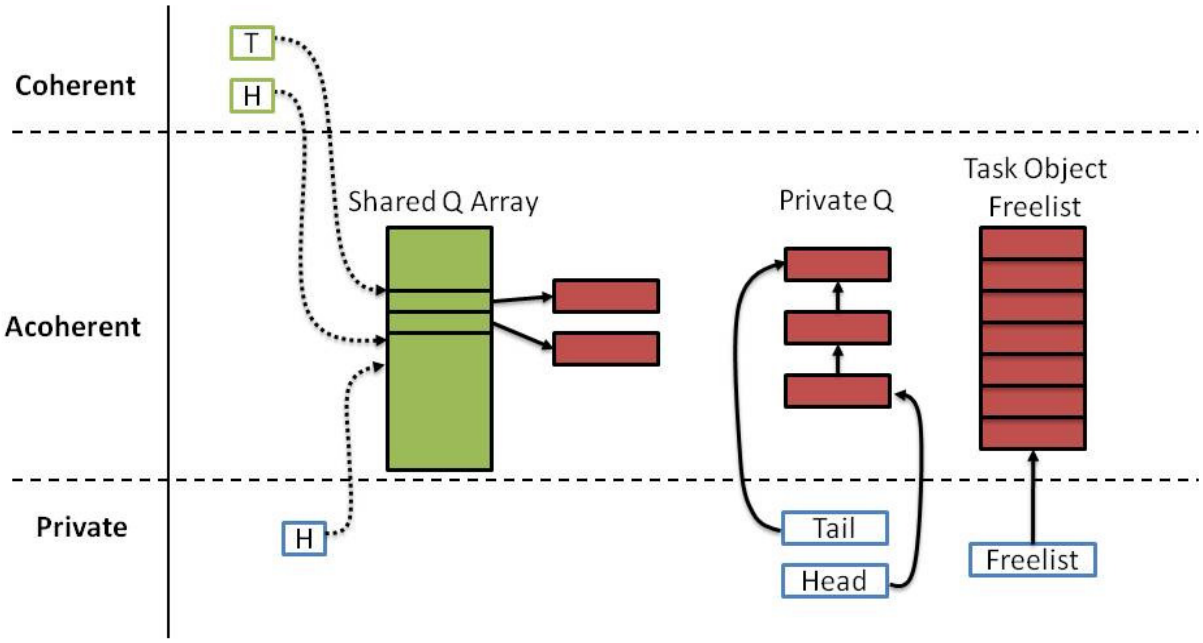

Figure 2-18. Structure of a single ASM task queue.

The shared and private queues are logically connected. Both the shared and private queues use task objects from the local freelist. If a remote thread steals from a local worker, it must create a copy of the task state using a task object from the thief's local freelist.

objects is a copy that occurs during stealing. We justify this additional overhead by noting that steals are inherently long latency operations anyway. Adding an additional object copy is unlikely to make a noticeable impact.

We show the ASM task queue implementation graphically in Figure 2-18. The queue data structure has elements spanning three types of segments. The local task object pool and the underlying array for the shared queue are both allocated into a single acoherent segment that, by convention, is owned by a single worker. The T and H indices for the THE protocol are allocated in coherent memory because they are fundamentally synchronization variables and are expected to accessed frequently by numerous processors (especially due to the queue partitioning). Finally, we allocate various pointers in a private segment, including pointers to the local freelist and private partition of the work queue. One additional private pointer is actually an index into

Owner Worker/Victim

```
01: push() {
02:   tmp = T;
03:   checkin(task pool);
04:   T = tmp + 1;
05: }
06:
07: pop() {
08:   T--;
09:   if (H > T) {
10:     T++;
11:     lock(L);
12:     T--;
13:     if (H > T) {
14:       T++;
15:       unlock(L);
16:       return FAIL;
17:     }
18:     unlock(L);
19:   }
20:   return OK;
21: }
22:
23: gc() {
24:   lock(L);
25:   local_H = H;
26:   unlock(L);
29:   while (private_H != local_H) {
30:       delete(private_H);
31:     private_H++;
32:   }
33: }
```

Remote Worker/Thief

```
34: steal() {
35:   lock(L);
36:   H++;
37:   if (H > T) {
38:     H--;
39:     unlock(L);
40:     return FAIL;
41:   }
42:   checkout(task pool);
43:   task* cpy = copy(H);
44:   unlock(L);
45:   return OK;
46: }
```

Figure 2-19. THE protocol implementation for ASM assuming the data layout in Figure 2-18.

the shared queue, and is used by the ASM version of THE to implement garbage collection on stolen task objects, as discussed in more detail below.

The shared queue is managed with an enhanced THE protocol that also takes care of segment management. We show the algorithm in Figure 2-19. The first difference from the reference implementation occurs on line 03, on which the owner checks in its insertion on the shared queue. The checkin occurs between the read and the write of the `T` index for reasons explained in the consistency Section 3.5.3. This `checkin` makes the new task visible to any workers that may try to steal it at a later time. Another difference is the use of a `checkout` on line 42. At this point,

the thief has successfully stolen a task but in order to see its contents the thief needs to check out the remote pool. If it didn't, the thief may observe stale task data. The thief also makes a copy of the task into an object allocated from the thief's own local task pool before returning. This is to maintain the invariant that workers only execute tasks from their local task pools.

Also notable about the enhanced THE protocol is what is *missing*. First, notice that the enhanced version removes the memory fences in both the **`push`** and **`steal`** routines. The fences are not needed because ASM guarantees sequential consistency for coherent accesses, which is explained in detail in Chapter 3. Second, notice that there is no **`checkin`** or **`checkout`** operation in the **`pop`** routine. The protocol avoids both operations because of the use of local task pools. When a worker pops from the queue it can be sure that no other worker has touched that task, and so there is no need to check out. Because the worker is consuming, not producing, data, there is also no need to check in. If workers were able to steal task objects from others (i.e., use non-local tasks), a **`checkout`** might be needed during a pop. By enforcing local task objects, though, we are effectively able to transfer any overhead from a checkout to the overhead required to copy during a steal. Again, because a steal is a rare and long latency operation, we assume it is better at hiding the additional overhead.

Owner Worker

```
01: read_messages() {
02:   checkout(msgQ.pool);
03:   if (msgQ.size > 0) {
04:     lock(msqQ.lock);
05:     checkout(msgQ.pool);
06:     while (msqQ.size > 0) {
07:       msg = pop(msqQ);
08:       <handle msg>
09:     }
10:   }
11: }
```

Thief Worker

```
12: send_message(msg) {
13:   lock(msgQ.lock);
14:   push(msgQ, msg);
15:   checkin(msqQ.pool);
16:   unlock(msgQ.lock);
17: }
```

Figure 2-20. Algorithm to manage message queues in ASM task runtime.

In addition the shared task queues, other interesting data structures in the task runtime are the message queues used to communicate information about stolen tasks. These queues are used primarily to inform a parent task that a stolen child has completed. While the message queues are the same abstract data type as the task queues (i.e., a double ended queue), they do not exhibit the same sharing characteristics and so we chose to implement them differently.

We expect the message queues to be modified rarely since messages are only generated in response to a steal. Also, the message queues have in effect the opposite sharing pattern of the task queue; messages are only consumed by the owner of the queue and messages are only produced by thieves. Because of these characteristics, we design the message queues so that remote updates are simple and don't expend great effort to optimize concurrent access (e.g., don't use a protocol like THE).

Messages are sent by thieves when a stolen task completes by pushing a message on the victim worker's message queue. Victims read the message queue by polling it between task switching events (e.g., at task sync time). We show pseudocode for the message queue routines in Figure 2-20. Unlike the task queues, *all* of the queue state is allocated from a single acoherent segment (not just the queue objects). Placing the entire data structure in a single segment makes it easy for multiple workers to modify the queue. We serialize access to the message queue push/pop routines with a basic lock, again to keep things simple and because contention is expected to be low. The lock only needs to be acquired by the owner if it finds messages in the queue (and thus needs to make a queue modification), allowing the owner to poll without the overhead of locking. Because the message queue is in an acoherent segment, though, the owner does need to check out the message queue segment before reading its state (line 02). Otherwise, it may never see new messages.

```
lock g_lock;       // global lock for STM runtime
int g_seq;         // global sequence number
queue<tx> g_tx_q;  // completed tx data; tx = tuple<rs, ws, begin_seq, end_seq>

tx my_tx;          // local tx data, allocated in private segment

begin_tx:
  g_lock.lock
  my_tx.begin_seq = g_seq++
  checkout(segment)
  g_lock.unlock

end_tx:
  my_tx.rs = lrs(segment); my_tx.ws = lws(segment);  // load my read/write sets
  g_lock.lock
  my_tx.end_seq = g_seq++
  foreach tx in g_tx_q
    if (overlap(my_tx.begin_seq, my_tx.end_seq, tx.begin_seq, tx.end_seq) and
       ( (intersect(my_tx.ws, tx.rs) or
          intersect(my_tx.ws, tx.ws) or
          intersect(my_tx.rs, tx.ws) ) )
        g_lock.unlock    // tx aborts
        checkout(segment)
        <STM cleanup>; return
  checkin(segment)       // commit tx
  g_tx_q.enqueue(my_tx)  // my_tx completed, add to g_tx_q
  g_lock.unlock
```

Figure 2-21 – Simple STM that uses ASM to efficiently perform version management. For simplicity, assume that transactions only work on one segment.

2.5.4 Case Study #4: Simple STM

We have shown that ASM can be used with existing programs; now we'll show that ASM can also be used to create new programs not possible with conventional cache coherent memory. Specifically, we will describe how ASM can be used to build a simple Software Transactional Memory (STM) runtime that exploits acoherence to achieve fast version management without explicit copy instructions.

Pseudo-code for the Simple STM runtime is shown in Figure 2-21. The runtime implements lazy conflict detection, eager version management (via acoherence), a simple earliest-committer-wins commit policy, and a global sequence number for transaction ordering [137]. For simplicity of exposition, we assume that transactions only modify one segment at a time. That requirement can be relaxed, but the details are outside of the scope of this dissertation.

Simple STM works by using checkout and checkin to manage versions of data in a transaction. When a transaction begins the transactional segment is checked out and when the transaction commits it is checked in. The runtime can also abort transactional updates by performing another checkout, which will restore the segment data. The semantics of checkout and checkin will ensure that transactional updates are not exposed prematurely. If the system Simple STM runs on does not support strong-acoherent segments (e.g., ASM-CMP, Chapter 4), Simple STM will also need to handle the possibility that a transaction causes an acoherence overflow exception. If this exceptional case occurs, an easy solution would be to grab a global lock and serialize transaction execution, similar to prior hybrid TM systems [50].

Simple STM also uses the ASM load read/write set instructions for conflict detection. Each transaction loads its own read/write set at commit time and compares those sets to those of any concurrent transactions. Read/write sets are shared through a special coherent segment to simplify the runtime.

2.6 Related Work

Generally there are two broad categories of programming models for multiprocessor systems: shared memory and message passing. ASM is most accurately categorized as a hybrid between the two. Like shared memory ASM uses a single unified address space, but like message passing requires software to explicitly declare communication points. From a programmability standpoint we think this achieves a good balance between the strengths of both. The unified address space makes pointer manipulation easy, allowing software to construct pointer-based data structures like lists with relative ease. On the other hand, by explicitly identifying communication, the ASM model gains some benefits of message passing safety (i.e., a remote

thread cannot spontaneously affect local work). There is also a meta-benefit that by forcing programmers to explicitly identify communication, programmers must *think* about how a program is communicating, which has the potential to head off performance bugs earlier in the design stage.

In this subsection we discuss prior work as it relates to the ASM programming model. We defer discussion of any implementation related prior work until Chapter 4 and work related to the subtleties of ASM consistency to Chapter 3. First we discuss shared memory models and then cover models most accurately described as message passing. Some related work, like ASM, is difficult to pin solidly as either shared memory or message passing; in such cases we choose the category we believe to be the closest conceptual match.

2.6.1 **Shared Memory**

The distinguishing feature of a shared memory programming model is a single address space. In a shared memory model a named location uniquely identifies a position in memory regardless of whom the requestor is or when the request was made. Shared memory facilitates fine grained sharing and allows, for example, programs to easily share small portions of a large structure without needing to marshal data into send/receive buffers. Perhaps for these reasons, shared memory has emerged as the *de facto* standard for small-to-medium size parallel computers, including nearly all chip multiprocessors (with the exception of some DSP and/or graphics chips [64,99]).

Because shared memory exposes a location to multiple processors at the same time, a shared memory model must implement a policy that determines when updates are seen by others. Shared memory models can be distinguished by amount and type of responsibility they place on software to enforce that policy. The most common policy is coherent shared memory. A system

providing coherence maintains the invariant that at any time there can be either multiple read-only observers or a single read-write observer, but never both. By allowing only a single read-write observer, the coherent invariant ensures that updates are (logically) propagated immediately to subsequent readers. Coherent shared memory can be implemented either in hardware, in a software runtime layer, or a hybrid of the two. We will discuss each variant in turn.

Hardware Cache Coherence. In systems implementing hardware cache coherence, software is absolved of all responsibility for maintaining the coherence invariant. Most modern commodity multiprocessor systems, including all versions of the ARM, MIPS, SPARC, and x86 architectures, support hardware cache coherence. From a programming standpoint it places the least burden (compared to the other variants of coherence) on developers for achieving functional correctness. However, because hardware cache coherence is, by design, an opaque mechanism to software, achieving optimal performance can be cumbersome and often requires significant ingenuity [1,66,71,113,162,167]. Furthermore, what leads to optimal performance on one implementation of hardware cache coherence may result in poor performance in a different implementation. We expand much more on this topic, and how it relates to ASM, in Chapter 4.

The ASM model has the potential to avoid the performance pitfalls of hardware cache coherence for at least two reasons. First, because ASM communicates semantic information about data to hardware, implementations can choose memory management policies that will perform optimally for a given data location. Second, because ASM exposes some of the underlying mechanisms required for data sharing, it makes it more likely that programmers will consider the performance implications of their code early in the design stage.

Systems implementing hardware cache coherence often make use of data prefetching to overcome some of the performance limitations. Prefetching can be directed either by hardware or by software. Hardware prefetchers attempt to predict regular (e.g., sequential or strided) access patterns and issue speculative requests for data it expects the program to need soon. Hardware prefetchers help hide the latency of large capacity memories in the memory hierechy, and can be used in conjunction with ASM to fulfill that same purpose. Software directed prefetching usually makes use of special prefetch instructions that are inserted either by a programmer or a compiler. Prefetch instructions are similar to acoherence operations; they effectively communicate to the hardware information about what software is likely to do next. The checkout operation, for example, is an indication to hardware that software is about to access data from the checked out segment.

ASM borrows intellectual insights and naming conventions from Cooperative Shared Memory [17]. Unlike CSM, though, the ASM check-out and check-in operations apply to an entire segment at a time (rather than a cache block) and are required for correctness (rather than a hint to hardware). ASM also shares some similarities to Tempest and Typhoon [30], as both can be thought of as a bridge between hardware shared memory and message passing.

Software Cache Coherence. To overcome the drawbacks of hardware coherence and to reduce hardware design costs, there have been many proposals to implement coherence in a software runtime layer. At a high level, ASM resembles many of these systems in that it requires some software intervention for correct behavior (i.e., **`checkout`** and **`checkin`**). ASM also builds on the foundations laid out by these prior systems related to consistency and semantic differentiation.

ASM has many similarities to the Munin Software Distributed Shared Memory (SDSM) system [20]. Applications using Munin annotate all shared objects based on an access pattern, and the underlying SDSM system uses a different coherence protocol for each type that is optimized for the particular pattern. ASM software also identifies different data types, but does so indirectly by allocating objects from different segment types. ASM also uses fewer access pattern types (four vs. seven) because many distinctions Munin makes, e.g., Migratory vs. Producer-Consumer, benefit from the same acoherent optimization. Munin was the first system to identify many of the characteristics that make ASM competitive with conventional coherence, including the benefits of delayed invalidation and multiple writers [34].

The ADSM system by Monnerat and Bianchini also differentiates data based on sharing type [134]. Their system automatically characterizes data as falsely shared, migratory shared, or producer/consumer shared, and applies the corresponding management policy that will optimize efficiency. ASM relies on software to communicate those patterns to hardware via **`checkout`** and **`checkin`** operations, and does not suffer from false sharing.

The C Region Library (CRL) by Johnson, et al. implements software coherence by breaking shared memory into regions and by controlling access at the region granularity [96]. For example, a thread can create a region, acquire or release read-only permission on a region, or acquire or release read-write permission on a region. CRL regions are similar to ASM segments in that they serve as the basic unit of sharing in the system. ASM segments use acoherence for consistency, however, rather than requiring software to explicitly control access through acquire/release calls. In comparison, we believe acoherence can lead to higher performing implementations because the acoherence semantics allow a region to read-write at multiple threads simultaneously.

Hybrid Coherence. Systems with scratchpad memories (e.g., the IBM Cell [98]) expose a small, software-managed RAM to each processor in addition to a single coherent global memory. While the scratchpad layout allows software to utilize private storage like ASM, the software management overhead of scratchpads can be significant. Scratchpads are managed at the block granularity, so software is forced to micromanage data movement much like in the incoherent model. Also, scratchpad memories have their own address space, complicating data movement and comparisons in software. For these reasons, scratchpad memories have been slow to gain widespread acceptance.

The Partitioned Global Address Space (PGAS) model has recently become popular in large high performance computing systems. The PGAS model contains a global memory space that is kept coherent among all observers. The global space is partitioned so that certain address ranges have an affinity for certain processors. The affinity especially helps in systems at the scale of high performance computing where locality is critical. In addition to the global memory, each processor has access to a local memory that is in a separate address space from global memory. Local memory is only observable by one processor. Like ASM, the local memory is abstracted so that software does not have to manage cached local memory (i.e., it is not a scratchpad). Unlike ASM, accesses to global memory are still fully coherent. Many variations of the PGAS model exist, including those implemented by the languages Chapel [39], Fortress [147], UPC [33], and x10 [41]. Predecessor languages also shared some of the PGAS properties, including Split-C [47] and Cilk [26].

Incoherence. Some shared memory programming models do not use a coherence layer at all and instead rely on application software to manage caches directly. Such a model is commonly found in multicore DSP architectures and other application specific designs. The Texas

Instruments C64x line, for example, provides a two level cache at each core but no automatic mechanisms for coherence [165]. Instead, the architecture provides instructions that can invalidate or writeback cache data. In the case of the C64x line, software has the option of applying those operations either on an individual cache line or globally to the entire cache (though other designs, e.g., the FreeScale MSC8144 [64], can apply the operation on a contiguous region of addresses). Software is responsible for ensuring that any shared data is correctly written back before it is consumed remotely (or ensuring that the data is uncached to begin with).

While the DSP software managed caching mechanism is outwardly similar to ASM, it differs in key ways that makes it a more feasible model for more complicated and irregular software than is typically found in a DSP chip. In ASM, the caches are still abstracted. Software does not need to know the specific size and/or associativity of a cache in order to apply a **`checkout`** or **`checkin`** operation. Also, the ASM operations implicitly track cache usage, absolving software of the responsibility to remember which addresses need attention at a **`checkout/checkin`** operation.

We liken the difference between acoherence and software managed caches to garbage collection vs. manual memory management. While determining when and what allocations need to be freed during the course of an execution initially seems like a simple task, in practice it is quite difficult, thus the existence garbage collectors. Likewise, determining which blocks need to be written out of a cache, especially taking into consideration evictions over time, can prove to be an arduous task for software, thus the benefit of acoherence.

Some previously proposed models, like ASM, give software the ability to change how shared memory is managed for different locations. Rigel gives software the ability to mark cache block-

sized regions as either coherent or incoherent depending on whether or not that data would benefit from hardware coherence [95,103]. For incoherent data the Rigel system also provides primitives similar to those found in DSP architectures for maintaining coherence in software. Other proposals differentiate data automatically, for example by predicting that a location on the stack should be private [114] or that a location has strong temporal or spatial (but not both) locality [74,97]. While these models are similar to ASM in that they allow different semantics for different data, none use the acoherent model as an option.

DeNovo is a hardware architecture designed to reduce the complexity of cache coherence by relying on help from software [42]. The architecture exploits the growing trend in parallel programming languages that enforce strict, disciplined, memory sharing by propagating semantic information from the language level down to hardware. The underlying coherence protocol in a DeNovo system can take advantage of such information (e.g., the program is data race free) to eliminate significant complexity and overheads. Like DeNovo, ASM also relies on software to pass information to hardware, but does so at a different layer (i.e., does not rely on compiler analysis) and also uses acoherence rather than simplified coherence.

The Intel Cloud on a Chip research processor is a 48 core single chip multiprocessor [83] that uses incoherent caches. The primary communication mechanism is a message passing library, but the designers also chose to support a limited amount of software managed cache coherence. Like ASM, the Cloud on a Chip uses a special segment of virtual memory to distinguish between variables that are private (message passing) and shared (software coherence). Unlike ASM, though, the shared region is not acoherent, but is instead something resembling release consistent with explicit ownership transfers (detailed descriptions of the mechanism are not yet available as of this writing).

Transactional Memory. In 1993 Herlihy and Moss proposed the idea of transactional memory as a way to implement complex synchronization operations [85]. Over the next several decades the idea gained great traction and eventually grew into a programming model in its own right [82]. Memory operations within a transaction are guaranteed to be atomic and isolated with respect to other threads in the system and, notably, are guaranteed to complete. Variations differ regarding the relationship between transactional and non-transactional operations, with some enforcing atomicity and isolation even if a non-transactional operation interferes with a transactional operation [11,19,43,121,130,137,148], others that break under non-transactional interference [50,117,123,144,153,156,157], and one that disallows non-transactional operations altogether [81]. All, however, incur complexities not found in ASM. Transactional memory systems must perform three basic tasks: version management, conflict detection, and conflict resolution. In comparison, ASM only handles version management (and in the case of best-effort/weak acoherence, does so in a simplified manner). The remaining two tasks are sources of significant complexity. Conflict resolution, especially, can be tricky to deal with because the correct resolution policy is workload (and even phase) dependent [27,72] and because it nominally requires support for speculative execution. Because of the complicated decisions made during conflict resolution, we believe it is best left under software control, which it would be in a software TM system built on ASM.

GPU Models. There has been a recent surge of interest in GPU architectures because of their rising use as general purpose processors [100]. GPUs have a memory model that differs significantly from commodity architectures largely because they make very little attempt to abstract away the details of the physical memory layout [84]. In part because software is forced to explicitly program physical memory movement (and thereby provide tremendous semantic

information to hardware), the GPU memory model has the potential to lead to highly efficient execution. The trade off, of course, is that the burden of software can be quite high. Like GPU memory models, ASM is a model that is closer to the physical layout of most memory hierarchies than flat coherent memory, but still abstracts the low level details enough so that, for example, the same software can be expected to run well on different ASM implementations.

Miscellaneous. Recently there have been many influential proposals that, like ASM, suggest breaking software computability in the name of performance, efficiency, and/or programmability. The Singularity project championed by Hunt and Larus [110,111] proposes a vastly redesigned operating system interface for large multiprocessor systems. Relevant to ASM is the use of the exchange heap for thread (or, in Singularity parlance, Software-Isolated Processes) communication. Threads can only communicate through the exchange heap and all other memory is considered private. Access to the exchange heap is tightly controlled to ensure mutual exclusion.

Ford's Determinator kernel is an operating system designed for deterministic parallel processing [14]. Any application executed in the system is guaranteed to be exactly repeatable. The kernel achieves this property by enforcing a fork/join style of execution and by tightly controlling how parent processes can access results from their children. Notably, this means that applications run in the Determinator system do not use conventional shared memory.

2.6.2 Message Passing

The message passing model is generally distinguished by the presence of disjoint address spaces. To communicate, software is usually given mechanisms to transfer data between address spaces. Message passing models are usually easier to reason about than shared memory programs because all observable effects are local, i.e., remote processor cannot cause updates

spontaneously to a local location. As a result, message passing programs have less nondeterminism, are easier to debug, and are easier to performance tune.

The most common message passing models are MPI and PVM [77]. Both use send/receive primitives for communication. Send and receive messages are paired, i.e., when software sends a message it must explicitly specify a receiver and when it receives a message it must explicitly identify the sender. ASM is similar in that it requires software to identify all communication points but differs in that **`checkout`** and **`checkin`** are not explicitly paired. Of course, ASM also supports a single address space.

The Linda system is a runtime kernel that implements messaging primitives for shared memory programs [8]. Linda provides three basic operations: **`out`**, that puts a tuple into tuple space, **`in`**, which removes a tuple from tuple space and returns it, and **`read`**, that returns a tuple from tuple space but does not remove it. An ASM **`checkin`** is similar to a (confusingly named) Linda **`out`** operation and a **`checkout`** is similar to an **`in`**. They differ, however, in several ways. First, the tuple space is not in the same address space as local memory, in contrast to ASM in which all of memory shares a unified address space. Second, the **`in`** and **`out`** operations only apply to tuples at a time, which roughly correspond to individual data objects; ASM, on the other hand, applies checkout and checkin operations to segments at a time that may contain multiple data structures. Third, the Linda system will block on an **`in`** operation if a tuple by that name does not exist while an ASM checkout will never block. One way to think about Linda (indeed one way to use it) is as an abstracted form of a parallel task queue. The same analogy is not applicable to ASM.

Ramachandran and Khalidi proposed a shared memory architecture that used explicit **`page_in`** and **`page_out`** operations on memory segments [149]. **`page_in`** and **`page_out`** operate similarly

to the Linda in and out operations, but rather than applying to a tuple they apply to a memory segment like ASM. Unlike ASM, though, segments were also used as the basis for synchronization. Their system disallowed multiple writers at the same time, and provided a built in mechanism to lock down a segment. Also, like Linda, the **`page_in`** and **`page_out`** cause a consumer to block on a producer.

The Bulk Synchronous Parallel model [170], which serves as the foundation of many follow-ons [21,53,90,95], is a programming model that divides execution into global epochs. Within an epoch each processor operates on private memory. All communication occurs at epoch boundaries by sending messages. BSP is similar to ASM in that threads are given a private storage area, but BSP is more limiting than ASM. Because of the barrier-like nature of BSP, it is most applicable to data-parallel type programs and can be an awkward fit for applications that use more irregular algorithms. ASM, on the other hand, gives software more flexibility by allowing segment mappings, **`checkouts`**, and **`checkins`** to occur at arbitrary and local granularities.

3

ASM Memory Consistency

While the informal description of the ASM memory model presented in Chapter 2 is helpful to gain a working intuition of ASM operation, it is insufficient for a rigorous analysis. In this chapter we formally define the ASM memory consistency model. By formalizing the model, we give hardware designers an exacting reference to verify correct implementations, programmers a well-defined set of rules for communicating through shared memory, and researchers a framework in which to compare ASM to other memory models.

In Section 3.1 we motivate the need for a formal memory model and provide some background information and common terminology that will be used throughout this chapter. The section is not intended to be a complete overview of consistency, and we refer interested readers to excellent books and articles on the subject ([2,48,160]). In Section 3.2 we introduce the abstract system model used in our formalisms and the associated definitions and notation. Then we formally define the ASM memory consistency model in a hardware-centric manner in Section 3.4 and analyze it in Section 3.5. In Section 3.6 we introduce a subclass of executions for

which ASM is sequentially consistent. In Section 3.7, we show that ASM is sequentially consistent for the important class of data-race-free programs, which allows programmers to reason in terms of sequential consistency for most extant programming models [3]. Finally, in Section 3.10, we compare the ASM consistency model to other well-known models so that the familiar readers might gain a point of reference.

3.1 Motivation and Background

Informal specifications of memory semantics given in English prose are prone to error and misinterpretation [7]. This is especially true for ASM, which is new and unusual in many respects compared to prior art. For example, given the description in Chapter 2, it is difficult to discern whether or not ASM hardware must make checkout and checkin actions atomic (it does not).

A *memory consistency model* (a.k.a. a *consistency model* or a *memory model*) is a contract between hardware and software that defines which stored values load operations are allowed to observe. Consistency models are specified abstractly, without mention of a specific physical memory layout, and are thus applicable to a wide range of implementations. We define one model to be *weaker* than another if it allows an execution another does not and *stronger* if the converse is true. A model is *equivalent* to another if all valid executions of a program in the model are also valid in the other and vice versa. The relationship between models is important for both hardware designers and programmers alike; hardware designers are free to implement a system that provides equivalent or stronger consistency than the architecture defines and programmers can write portable code as long as it is correct for the weakest possible target architecture.

The most intuitive consistency model, and indeed the one implicitly assumed by many programmers [160], is Sequential Consistency (SC). An execution is sequentially consistent if the order of operations in a multiprocessor system is consistent with an identical execution on a multitasking uniprocessor, i.e., there is a single, total, observable order of all memory accesses [88]. SC is intuitive for programmers because it allows them to reason about a multiprocessor execution serially.

The class of models that are weaker than SC are called relaxed memory models, and include almost all commercially relevant systems [12,46,94,176]. Relaxed models are usually created in the name of performance [3,46,56,70,94,101,159,176], as SC compliant hardware must either insert frequent delays to enforce ordering or implement speculation mechanisms that can detect and correct violations of SC [73,181]. The ASM model gives programmers the ability to use private storage to achieve isolation if they wish and allows them to fall back on SC if not.

3.1.1 Two Frameworks

Many frameworks exist for formalizing memory models. Some, like the functional description of the SPARC models by Weaver and Germond [176], are hardware-centric and are good to use for rigorous formal comparisons to other models and as a way to gain insights into the relaxations a hardware implementation can perform. Others, like the class of *Sequential Consistency Normal Form* models proposed by Adve [5], are programmer-centric and relieve programmers of the details of relaxed models altogether as long as they create data-race-free code. Programmer-centric models have the downside that they leave behavior undefined in the presence of data races. Still others, like the x86-TSO model [155], are described operationally in a manner that is easy for programmers (especially those using imperative languages) to reason about and that are rigorously defined even in the presence of data races.

In this chapter we will explore the ASM model through two of the frameworks described above, hardware-centric and programmer-centric, and will prove that they are both equally valid. In Appendix A we also describe a third, operational description of the model but do not prove its equality to the first two. By having multiple descriptions, hardware designers, programmers, and researchers can choose the framework most amenable to their needs.

First, we formalize ASM in the hardware-centric manner by describing, given an execution, whether or not that execution is valid in the ASM model. We use this first formalization as a foundation to analyze the properties of ASM. In particular, we show that any execution that is both properly paired and lossless (as defined in Section 3.6) is compatible with sequential consistency. Second, we use the functional foundation to derive the data-race-free constraints in an ASM system, and show that all data-race-free programs execute in a sequentially consistent manner.

3.2 System Model and Definitions

This section introduces the abstract system model, definitions, and notation used in the formal memory model specification. The definitions in this section are in English. When appropriate, we define some terms more formally for the proof in Section 3.6. Importantly, the system model defined in this section does not correspond to any real physical machine. Rather, it is an abstract framework used to make the formal specification and associated analysis easier. We will show in Chapter 4 how a real machine might emulate the properties of the abstract system.

3.2.1 Definitions

Processor: A collection of states including but not limited to a program counter and a set of registers.

System: A collection of processors together with a shared set of memory locations. Note that, in this definition, memory locations do not hold values, but only serve as a common naming system for processors to communicate.

Instruction: A tuple <R, V> containing a rule R that deterministically and atomically transforms a processor from one state to another and a value V that is set if part of the instruction rule interacts with memory[6]. Unless otherwise noted, assume that an instruction refers to a dynamic instruction.

Memory Operation: A *Store* is an atom of an instruction rule that produces a value for a specific memory location. A *Load* is an atom of an instruction rule that observes the value of a specific memory location previously produced by a store. *Checkout* and *Checkin* are instruction rule atoms that affect which values subsequent loads will observe. Notably, for purposes of the model, checkout and checkin *have no associated actions*, and only serve as ordering markers.

Memory Transaction: Memory operations may be combined into transactions that appear to execute indivisibly with respect to the system as a whole (i.e., no other memory operation from any processor can come between them). Implementations of the ASM model must provide at least two transactions, namely $Load_{sync}$ and $Store_{sync}$. $Load_{sync}$ consists of a Checkout followed by a Load. Similarly, a $Store_{sync}$ consists of a Store followed by a Checkin. Implementations may provide additional transaction types that build on the two base types, e.g., an RMW_{sync} that atomically executes a $Load_{sync}$ followed by a $Store_{sync}$.

[6] Our definition here assumes that an instruction produces (observes) a *single* value. The model can be extended to architectures that produce (observe) more than one value as long as those instructions can logically be broken into disjoint pieces that affect at most one location (i.e., micro-ops). For example, a single x86 instruction loads a value, adds it to an accumulator, then stores the result can be broken into three micro-ops that load a value, add it to an accumulator, and store a value, respectively.

Program Thread: A collection of static instruction rules, with a well-defined start and end rules, which defines a task for a processor.[7]

Program: The collection of program threads.

Segment: A set of memory locations.

Execution: A partially-ordered sequence of instructions that results from one specific (of many possible) run of a program. An execution cannot be determined *a priori*, but is rather constructed as a program runs.

Conflict: Two load or store memory operations conflict if they are to the same address, at least one is a store, and the operations are executed by different threads.

Memory Model: A set of rules that constrain the order and value of memory operations. The memory model defines executions which are *not* valid, but does not specify, for a given program, what the result will be.

Program Order: A relation describing the ordering constraints of instructions due to instruction rules. Program order is a partial order because it does not order instructions from different processors, but any subset of program order that contains instructions from a single processor is a total order.

Memory Order: A global total order of memory operations performed by all processors in the system. Memory order itself does not itself imply the value of a load or store, i.e., the value of a load is not necessarily the value of the previous store in memory order. Unlike program

[7] We assume that a program thread includes the entire software stack that runs on a processor. For simplicity, though, the reader may think of a thread as a single user thread that runs without interruption. If context switching is properly performed (Section 2.3), the two are equivalent.

order, the memory order is not necessarily observable by any processor, but is rather constructed based on inferences from an observed execution.

Value Order: A global partial order of memory operations that respects causality among instruction values. In the value order, a load is ordered after a store if it observes the store's value. Operations of other types (i.e., (Store, Store), (Load, Store), (Load, Load)) are not ordered in value order.

Table 3-1. Notation for ASM consistency

Notation	Meaning
$L_i^s a$	Load from address *a* in segment *s* performed by processor *i*
$S_i^s a$	Store to address *a* in segment *s* performed by processor *i*
$X_i^s a$	Load or Store to address *a* in segment *s* performed by processor *i*
CO_i^s	Checkout segment *s* performed by processor *i*
CI_i^s	Checkin segment *s* performed by processor *i*
CX_i^S	Checkout or Checkin segment *s* performed by processor *i*
$Value(X)$	If X is a store, the value the store produces. If X is a load, the value the load observes.
$\overset{p}{\rightarrow}$	Program order
$\overset{p}{\rightarrow}_i$	Subset of program order containing instructions from processor i. Formally, $\overset{p}{\rightarrow}_i = \{(X_j, X_k) \in \overset{p}{\rightarrow} : i = j = k\}$
$\overset{m}{\rightarrow}$	Memory order
$\overset{v}{\rightarrow}$	Value order
$X <_p X'$	Operation X comes before operation X' in program order
$X <_m X'$	Operation X comes before operation X' in memory order
$X <_v X'$	Operation X comes before operation X' in value order
$max_{\overset{x}{\rightarrow}}(T)$	An operation of type T that appears latest in total order $\overset{x}{\rightarrow}$
$max_{\overset{x}{\rightarrow}}(T, X)$	An operation of type T that appears latest in total order $\overset{x}{\rightarrow}$ and before operation X
$next_{\overset{x}{\rightarrow}}(T, X)$	An operation of type T that appears after operation X in total order $\overset{x}{\rightarrow}$
$\mathbb{E}$	An execution Formally, the irreflexive transitive closure of program and value order: $\left(\overset{p}{\rightarrow} \cup \overset{v}{\rightarrow}\right)^+$

3.2.2 Notation

We define the ASM model in terms of load, store, atomic, checkout, and checkin operations. Table 3-1 lists the notation used to represent those operations. If a term is missing, it is assumed to mean *any* (e.g., L_i^s means a load from any address in segment *s* on processor *i*). In any equation, an overbar is used to distinguish between two operations that would otherwise have the same notation (e.g., S_i^s and $\overline{S_i^s}$ are two distinct stores that otherwise have the same properties).

3.3 Two Existing Models

In this subsection we formally define two existing consistency models, namely Sequential Consistency (SC) and Total Store Order (TSO). We do so for two reasons. First, the formalisms should help orient readers familiar with the existing models to our notation. Second, we use the definitions later to show that (a) the ASM model is equivalent to SC for well formed programs, and (b) that the ASM model is similar to TSO despite a seemingly large conceptual gap.

3.3.1 Sequential Consistency

As defined by Lamport [108], a system is sequentially consistent if:

> *"…the result of any execution is the same as if the operations of all the processors were executed in some sequential order, and the operations of each individual processor appear in this sequence in the order specified by its programs."*

To enable an easier comparison with the ASM model, and to avoid ambiguities due to English prose, we now formalize SC using the same notation used for ASM. We include checkout and checkin operations in the formalization. Because checkout and checkin do not appear in the SC value equation, they are effectively no-ops in this formalization (but will become relevant later).

$$X_i <_p \overline{X_i} \Rightarrow X_i <_m \overline{X_i} \quad (3\text{-}1)$$

$$CX_i <_p \overline{CX_i} \Rightarrow CX_i <_m \overline{CX_i} \quad (3\text{-}2)$$

Figure 3-1. Memory order constraints for SC

$$\text{value}(La) = \text{value}\left(\max_{\underline{m}} \left(Sa \mid Sa <_m La\right)\right) \quad (3\text{-}3)$$

Figure 3-2. Value of memory operations in SC

Sequential consistency essentially says that there is a total order of all memory operations in the system that is observable by all threads. As the equations in Figure 3-1 show, if an operation *A* appears before operation *B* in program order, *A* must also appear before *B* in memory (i.e., SC *respects* program order). In SC, the value of a load is always the value of the store to the same address most recent in memory order, as mandated by the value equation in Figure 3-2. Consequently, whenever a value (logically) becomes visible to one processor, it must also (logically) be visible to all other processors.

3.3.2 Total Store Order

Total Store Order is a model originally created by the SPARC standard and that is widely suspected to be implemented by all x86 architectures [155,160,176]. TSO relaxes sequential consistency by allowing a processor to observe its own store before it is visible to the system as a whole (i.e., it allows local bypassing out of a private cache such as a store buffer). So that programmers can still make sense of their applications, TSO uses memory fence instructions that, after completing, guarantee that all previous stores are globally visible in the system.

We define TSO formally using our notation in Figure 3-3 and Figure 3-4. The first three memory ordering constraints, (3-4), (3-5), and (3-6), are the component parts of the SC constraint (3-1) minus Store->Load ordering. By omitting Store->Load ordering, the model allows a load to appear in memory order before a store later in program order. Combined with the TSO value equation, (3-10), this reordering allows a processor to see its own write early.

$$L_i^S <_p \overline{L_i^S} \Rightarrow L_i^S <_m \overline{L_i^S} \quad \text{\# Load -> Load} \quad (3\text{-}4)$$

$$L_i^S <_p S_i^S \Rightarrow L_i^S <_m S_i^S \quad \text{\# Load -> Store} \quad (3\text{-}5)$$

$$S_i^S <_p \overline{S_i^S} \Rightarrow S_i^S <_m \overline{S_i^S} \quad \text{\# Store -> Store} \quad (3\text{-}6)$$

$$S_i <_p FENCE_i \Rightarrow S_i <_m FENCE_i \quad \text{\# FENCE orders stores} \quad (3\text{-}7)$$

$$FENCE_i <_p L_i \Rightarrow FENCE_i <_m L_i \quad \text{\# FENCE orders loads} \quad (3\text{-}8)$$

$$FENCE_i <_p \overline{FENCE_i} \Rightarrow FENCE_i <_m \overline{FENCE_i} \quad \text{\# Total order of FENCEs} \quad (3\text{-}9)$$

Figure 3-3. Memory ordering constraints for TSO

$$\text{value}\left(L_i^S a\right) = \text{value}\left(\max_{\underline{m}}\left(S^S a \mid \quad S^S a <_m L_i^S a \quad or \quad S^S a <_p L_i^S a\right)\right) \qquad (3\text{-}10)$$

Figure 3-4. Value of memory operations in TSO

The value equations warrant further discussion, especially since, as we will show, it is also relevant to ASM. The value equation states that load will always observe the value of store that last in memory order that is either (a) before the load in memory order, (b) before the load in program order, or (c) both. There is no notion of "preference" to either local or global stores; a load always receives the value of the last store in memory order that satisfies those constraints, period.

If any store S is separated by any load L by a FENCE, then S must appear in memory order before L. In other words, the FENCE prevents bypassing before the store is globally visible. Globally, FENCEs are totally ordered (constraint (3-9)). Most implementations of TSO establish that total order implicitly by locally stalling a processor until all in flight stores have been ordered.

3.4 Functional Specification of ASM

Like the specifications above for SC and TSO, the model consists of two logical parts. First, the model defines constraints that determine a legal memory order for an execution. Second, the model defines *exactly* what value a load will observe based on its position in both memory order

$$L_i^S a <_p \overline{L_i^S a} \Rightarrow L_i^S a <_m \overline{L_i^S a} \quad \text{\# Load -> Load to same address} \quad (3\text{-}11)$$

$$L_i^S a <_p S_i^S a \Rightarrow L_i^S a <_m S_i^S a \quad \text{\# Load -> Store to same address} \quad (3\text{-}12)$$

$$S_i^S a <_p \overline{S_i^S a} \Rightarrow S_i^S a <_m \overline{S_i^S a} \quad \text{\# Store -> Store to same address} \quad (3\text{-}13)$$

$$S_i^S a <_p CI_i^S <_m CO_j^S <_p L_j^S a \Rightarrow S_i^S a <_m L_j^S a \quad \text{\# Paired CI-CO like distributed fence} \quad (3\text{-}14)$$

$$CX <_p \overline{CX} \Rightarrow CX <_m \overline{CX} \quad \text{\# Total order of CI/CO} \quad (3\text{-}15)$$

Figure 3-5. Common memory order constraints for ASM.

$$\text{value}\left(L_i^S a\right) = \text{value}\left(\max_{\underline{m}}\left(S^S a \mid \quad S^S a <_m L_i^S a \quad or \quad S^S a <_p L_i^S a\right)\right) \qquad (3\text{-}16)$$

Figure 3-6. Value of memory operations in weak acoherent, coherent and private segments.

($\underline{m}$) and program order ($\underrightarrow{p}$). Note that a single memory order is defined for all operations from all segments. Thus, even though different segment types may have different constraints on memory order, they are all constraints on the one and only memory order for an execution.

For simplicity, the memory model assumes that all loads and stores write exactly one word, though the model can be trivially extended to allow for other granularities. The model assumes that before execution begins, each processor performs a store to every memory location within a segment followed by a single checkin and checkout, and repeats for every segment. The value produced by the initial store is initial value of that location assumed by the program to be run. When a thread completes, it issues a final checkin to every segment.

3.4.1 Common Memory Order Constraints

There are a set of five constraints that apply to all ASM segment types, listed in Figure 3-5. The first three are like the first three constraints of TSO in Figure 3-3 except that they only apply to loads and stores to the same address. Like TSO, by omitting store to load ordering, loads can bypass early from a local store. Unlike TSO, and because ordering is only enforced per address, operations to different addresses do no need to appear in FIFO order (e.g., implementations can use coalescing buffers).

Table 3-2. Rules for weak acoherent segments

Memory Order Constraints	Common [(3-11), (3-12), (3-13), (3-14), (3-15)]
Value Equation	(3-16)
Implicit Operations	None
Undefined Behavior	Clobbering checkouts, i.e.: $\exists S_i, CO_i : S_i <_p CO <_p \text{next}_{\underline{p}}(CI, S_i)$

Constraint (3-14) is the ASM version of a memory fence. In contrast to TSO, which uses a single thread-local operation to order stores, the ASM fence is composed of a distributed checkin-checkout pair. Informally, the constraint states that a store is only guaranteed to be memory ordered before a load if those two operations are separated by an ordered checkin/checkout pair. In combination with the value equation of Figure 3-6, it guarantees that processors will see all globally checked in data after completing a checkout.

Finally, constraint (3-15) ensures that there is a total order of checkout and checkin operations, inclusive. This rule does not apply only to operations on a particular segment, but rather orders all checkout and checkin operations regardless of the segment they affect. Note that this does *not* mean that checkout and checkin actions need to complete atomically (more on this in Section 3.5.2).

3.4.2 Weak Acoherence

Weak acoherence does not add any additional memory ordering constraints above the common ones listed in Figure 3-5. With a valid memory order established, the load value equation in Figure 3-6 defines exactly which store a load will observe. Notice that the load value equation is the *exact same* equation that governs load values for TSO.

In weak acoherence, behavior is undefined if a processor performs a checkout between any store and the next checkin (i.e., checkout with unchecked-in updates in working memory). We

$$S_i^S <_p \text{next}_{\underline{p}}(CI^S, S_i^S) <_p \text{next}_{\underline{p}}(CO^S, S_i^S) \Rightarrow \text{next}_{\underline{p}}(CI^S, S_i^S) <_m S_i^S \quad (3\text{-}17)$$

$$S_i^S <_p CO <_p \text{next}_{\underline{p}}(CI, S_i^S) \Rightarrow S_i^S <_m \max_{\underline{p}}(CO^S, S_i^S) \quad (3\text{-}18)$$

Figure 3-7. Memory order conditions for strong and best-effort acoherence.

$$\text{Let } \text{firstL}(L_i^S a) = \min_{\underline{p}}\left(\overline{L_i^S a}, \max_{\underline{p}}(CO, L_i^S a)\right)$$

$$\text{Let } \hat{\text{S}} = \left\{S : \exists CI \mid S <_p CI <_p next(CO, S)\right\}$$

$$value\left(L_i^S a\right) = value\left(\max_{\underline{p}}\left(S_i^S a \mid \max_{\underline{p}}(CO^S, L_i^S a) <_p S_i^S a <_p L_i^S a\right)\right) \quad (3\text{-}19)$$

$$\text{or, if } S_i^S a \text{ does not exist,}$$

$$= value\left(\max_{\underline{m}}\left(\hat{S}_j^S a \mid \hat{S}_j^S a <_m \text{firstL}(L_i^S a)\right)\right)$$

Figure 3-8. Value equation for strong and best-effort acoherence.

chose to make this behavior undefined because we cannot construct an example where it would be useful for software to perform a clobbering checkout in a weak acoherent segment and because defining it would unnecessarily complicate the model equations (see strong acoherence below). If in the future there were a convincing argument for defining clobbering checkouts, we may choose to revise the model. We summarize the rules for weak acoherence in Table 3-2.

3.4.3 Strong Acoherence

Strong acoherence uses the common memory ordering constraints in Figure 3-5 plus two additional constraints shown in Figure 3-7 that govern where stores may appear relative their surrounding checkin and checkouts. Additionally, strong acoherence uses a different load value equation, shown in Figure 3-8 that further restricts which store will be observed by a load.

The strong acoherence definition, while syntactically complicated, is rather intuitive. The seemingly obscure constraints combine to enforce three key properties of strong acoherence:

1. Store Isolation. In strong acoherence, stores are not visible to remote threads until they have been checked in. This property is enforced mainly by constraint (3-17), which ensures that

Table 3-3. Rules for strong acoherent segments

Memory Order Constraints	Common [(3-11), (3-12), (3-13), (3-14), (3-15)] + (3-17), (3-18)
Value Equation	(3-19)
Implicit Operations	None
Undefined Behavior	None

all "normal" stores are ordered *after* their publishing checkin, i.e., $S_i^S <_p CI_i^S \Rightarrow CI_i^S <_m S_i^S$. Constraint (3-17) looks more complicated than that because it specifically excludes from the rule stores that are clobbered by a checkout. Clobbered stores are governed instead by constraint (3-18), as explained next.

2. Clobbering Checkouts. In strong acoherence, any store that is preceded by a checkout before a checkin will be clobbered, such that its value is not visible to (a) any remote load, or (b) any local load that occurs after the clobbering checkout. This property is enforced by constraint (3-18) and the load value equation, which together effectively "sequester" a clobbered store in memory order so that it will never be observed. Clobbered stores must appear before the preceding (in program order) checkout (constraint (3-18)), and consequently cannot be observed by remote threads (value equation (3-19), because the set $\hat{S}$ excludes clobbered stores). Locally, the value equation only permits bypassing up to the most recent (with respect to the load) checkout.

3. Repeatable Reads. Strong acoherence guarantees isolation after first use (and up to the next checkin or checkout), and must, unlike weak acoherence, prohibit loads from observing different values. This is enforced entirely by the load value equation (3-19). First, if a local store exists it will always take priority over a remote store, guaranteeing isolation, and thus repeatable reads, once a local store is performed. The equation ensures repeatable reads in the

Table 3-4. Rules for best-effort acoherence segments

Memory Order Constraints	Common [(3-11), (3-12), (3-13), (3-14), (3-15)] + (3-17), (3-18)
Value Equation	(3-19)
Implicit Operations	A spontaneous checkout may occur between any two operations.
Undefined Behavior	None

absence of a local store by defining the value of any load in terms of the first load after the most recent checkout rather than in terms of the load itself. In doing so, it ensures that all loads will logically occur at first use time.

There are no undefined behaviors or implicit conversions in strong acoherence. In Table 3-3 we summarize the rules for strong acoherence.

3.4.4 Best-Effort Acoherence

The rules for best-effort acoherence are identical to strong acoherence with the addition of one rule that says an implicit checkout may occur between any two instructions. We summarize the rules for best-effort acoherence in Table 3-4.

3.4.5 Coherent Read-Write

Coherent read-write accesses follow the common memory ordering constraints in Figure 3-5 and the value equation in. Additionally, all coherent read-write loads are implicitly converted to $Load_{sync}$ operations and all coherent read-write stores are implicitly converted to $Store_{sync}$

Table 3-5. Rules for coherent read-write segments

Memory Order Constraints	Common [(3-11), (3-12), (3-13), (3-14), (3-15)]
Value Equation	(3-20) or (3-19) or (3-16) – All are equivalent
Implicit Operations	All stores are converted to $Store_{sync}$s All loads are converted to $Load_{sync}$s
Undefined Behavior	None

$$\text{value}\left(L_i^S a\right) = \text{value}\left(\max_m \left(S^S a \mid \quad S^S a <_m L^S a\right)\right) \qquad (3\text{-}20)$$

Figure 3-9. Load value equation for coherent segments.

Because of the constraints on coherent accesses, this equation is equivalent to equations (3-16) and (3-19).

operations. As we prove in Section 3.6.3, treating coherent accesses as synchronized makes them sequentially consistent with respect to one another. Because of this, all three value equations Figure 3-6, Figure 3-8 and Figure 3-9, are valid and equivalent for coherent read-write accesses. We present a different load value equation only for convenience. Table 3-5 summarizes the model for coherent read-write accesses.

3.4.6 Coherent Read-Only

Coherent read-only accesses follow the common memory ordering constraints in Figure 3-5. Stores to coherent read-only segments are treated as no-ops. Because stores are disallowed, loads will always observe the value of the initial store to that address. Thus, all of the previous value equations are valid and equivalent. Table 3-6 summarizes the rules for coherent read-only segments.

Table 3-6. Rules for coherent read-only segments

Memory Order Constraints	Common [(3-11), (3-12), (3-13), (3-14), (3-15)]
Value Equation	(3-20) or (3-19) or (3-16) – All are equivalent
Implicit Operations	Stores become no-ops
Undefined Behavior	None

Table 3-7. Rules for private segments

Memory Order Constraints	Common [(3-11), (3-12), (3-13), (3-14), (3-15)]
Value Equation	(3-16)
Implicit Operations	None
Undefined Behavior	Access by a non-owner thread

3.4.7 Private

Private accesses follow the common memory ordering constraints in Figure 3-5 and the value equation in Figure 3-6 (weak). Behavior is undefined for an access from the thread that is not the segment owner. Notably, checkout and checkin *are* still defined so that a thread can migrate from one processor to another.

3.4.8 Type Changing

The model also allows segment types to change dynamically, which allows, for example, a program to transform a coherent read-write segment that has been populated with initial values into a coherent read-only segment for the remainder of the execution. The change procedure is as follows:

1. All threads check the segment in.
2. One thread changes the segment type then broadcast a completion signal.
3. All threads check the segment out.

Behavior is undefined if any thread performs a load or store to the segment between the final checkin before the switch and the initial checkout after the switch.

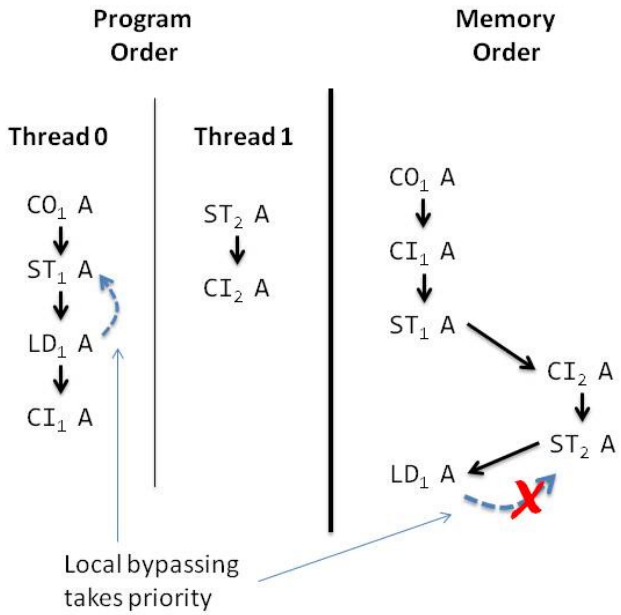

Figure 3-10. Stores are locally isolated in strong/best-effort acoherence.

3.5 Analysis

In this subsection we analyze the hardware-centric model presented above. First, in Section 3.5.1, we provide examples to show what the strong and best-effort constraints provide. In particular, we show examples that give insights into the isolation guarantee made by strong/best-effort acoherence and show how the constraints work together to guarantee clobbered stores are only read by local loads that are clobbered by the same checkout.

Second, in Section 3.5.2, we show some of the implications that the ASM model has for an implementation.

Third, in Section 3.5.3, we tie the constraints together in larger examples that demonstrate how the model guarantees the properties of ASM described in Chapter 2. In particular, show how mutual exclusion can work, why checkout and checkin actions do not need to appear atomic, and why consecutive checkins are generally safe for any segment type.

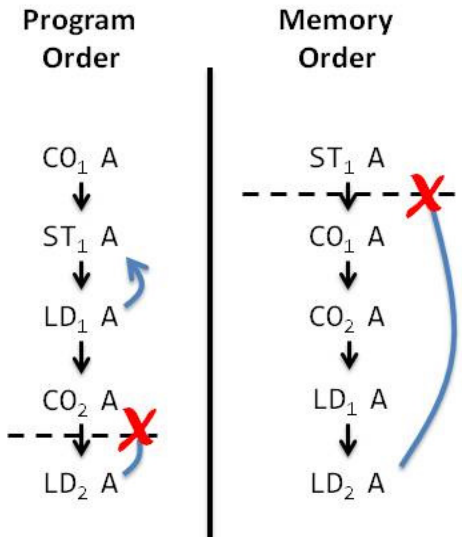

Figure 3-11. Local speculation in strong/best-effort acoherence. Load LD_2 can never observe store ST_1.

Finally, in Appendix A we provide the rationale for each common constraint by providing an example of bad behavior that would occur without it. Readers may pick and choose from these examples, focusing on any individual constraints that may not intuitively make sense.

3.5.1 Strong/Best-effort Constraints

In this subsection we show how the constraints for strong and best-effort acoherent segments work together to (a) ensure stores are isolated until checkin and (b) stores clobbered by a checkout can only observed by speculative local loads that also clobbered by the same checkout.

Because, for purposes of the model analysis, strong and best-effort are equivalent, we only refer to strong in the text below.

We show two examples to highlight the subtleties of isolation in strong acoherence. First, in Figure 3-10 we show an execution containing a load that will always read from the most recent local store regardless of memory order. Even though ST_2 comes after ST_1 in memory order (and both before LD_1), LD_1 does not get the value of ST_2 because the strong acoherence value equation gives explicit priority to local stores up to the most recent checkout. This example

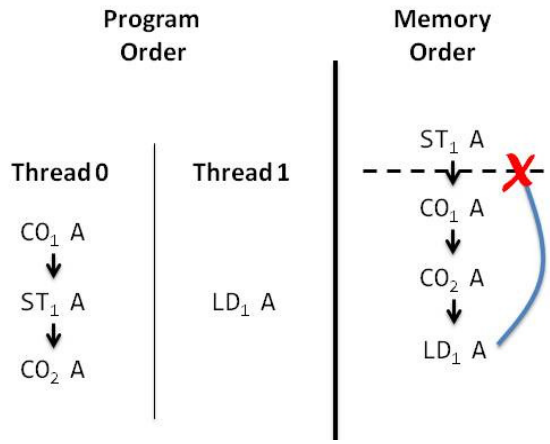

Figure 3-12. Global speculation support in strong/best-effort acoherence.

shows that after the first store, the value of a location will not change due to other (racey) stores in the system until the next checkout.

Next we show two examples highlighting how strong acoherence supports speculative execution. First, in Figure 3-11, we show how the model supports clobbering local stores. The second checkout, CO_2, clobbers store ST_1 because ST_1 is never checked in. Load LD_1, which comes before the clobbering checkout in program order, is the only load that can observe the value of ST_1. Load LD_2 cannot observe ST_1 by bypassing locally because the strong acoherence value equation prohibits bypassing past the most recent checkout (horizontal dashed line in program order). Load LD_2 also cannot observe ST_1 out of memory order because the value equation prohibits observing any store that is ordered before (horizontal dashed line in memory order) its most recent checkout (which constraint (3-18) makes true for all clobbered stores). Note that loads LD_1 and LD_2 could appear *anywhere* in memory order (as long as LD_1 still comes before LD_2, constraint (3-11)), and the result would still be the same.

The final example of strong acoherence, in Figure 3-12, shows how speculation is supported by the model when multiple threads are involved. We show a possible memory orders for an execution of threads 0 and 1 in which load LD_1 is ordered after store ST_1 in memory order. Like

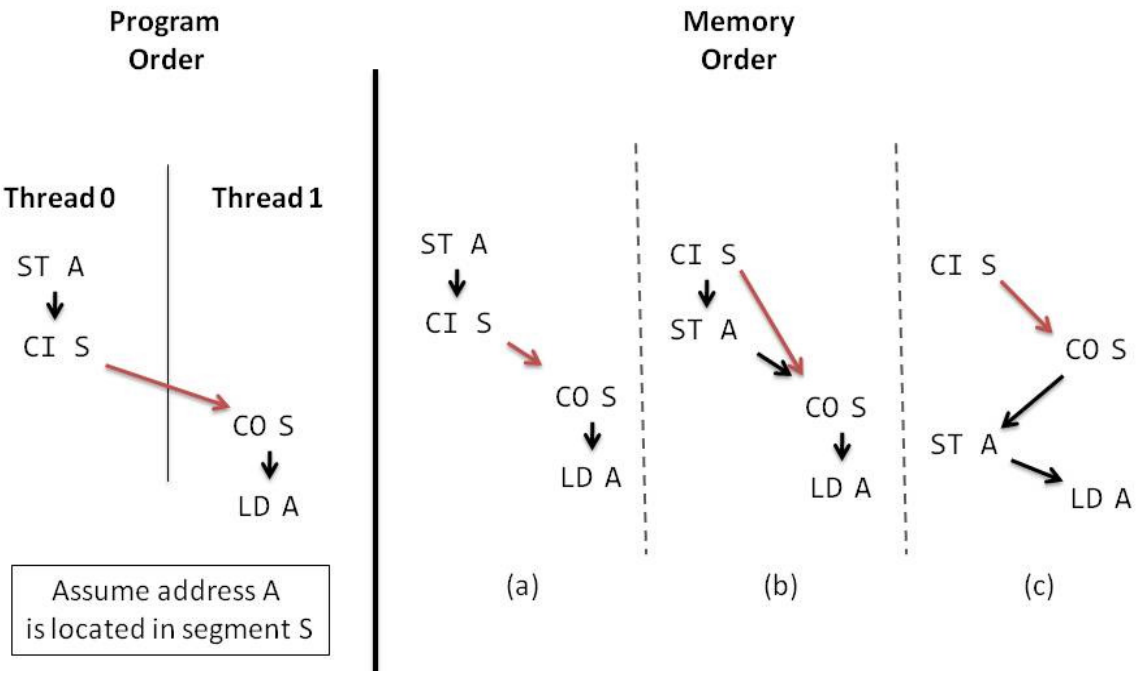

Figure 3-13. Stores and checkins are not atomic.

We show three possible memory orders for an execution of the program on the left, in which CI occurs before CO. Based on an informal discussion of ASM semantics, one might assume that (a) is the only valid memory order (as would be the case if checkin and checkout were more traditional fence instructions like release and acquire in release consistency). In actuality, the model allows ST to occur *anywhere* in memory order as long as it occurs before LD, making both (b) and (c) possible orders. Note all three orders are valid for weak acoherence. In strong acoherence, the "obvious" ordering (a) is actually illegal since stores cannot appear before they are checked in (this prohibits LD from seeing ST before it has been checked in).

the previous example, the combination of constraint (3-18) and the strong coherence value equation form a barrier (dashed horizontal line) that the load cannot see past. If the load instead appeared in memory before the store, then the value equation would again prohibit it from receiving the value of ST_1. Thus, in strong acoherence, clobbered stores are never visible to remote loads.

3.5.2 Implementation Implications

In this subsection we show highlight three implications of the model as they relate to a potential implementation.

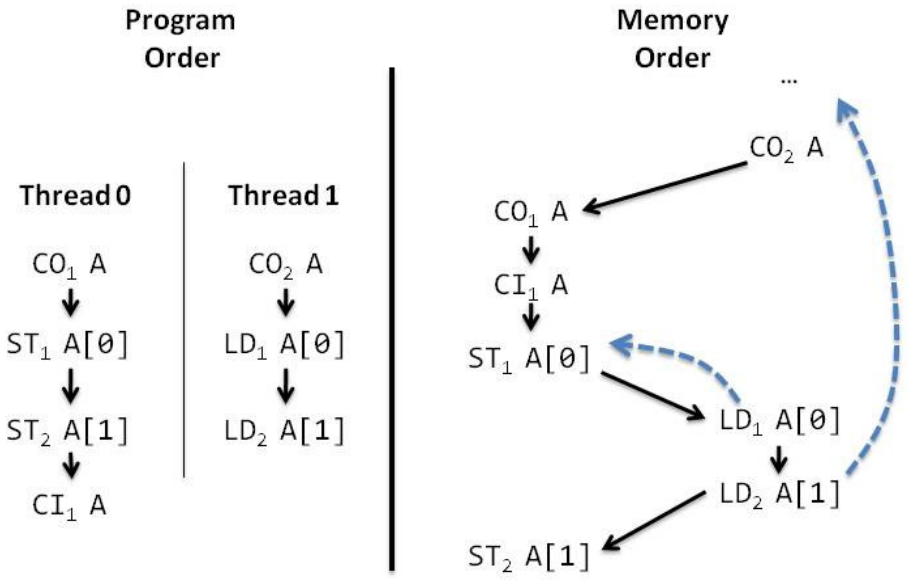

Figure 3-14. Checkin is not atomic

Checkin Actions are not atomic. One attribute of ASM that becomes clear in the consistency formalization is that checkout and checkin operations do not need to be atomic, even in strong acoherence. For example, an implementation does not need to wait for all previous stores to be globally visible before ordering a checkin. Consider equation (3-14), which ensures that any checkin orders prior stores before loads following a later checkout. This rule does not order stores to other stores or stores to their associated checkin. Thus, in memory order, stores and checkins can appear in any (non-atomic) order as long as all of the stores occur before any dependent (determined by checkin-checkout order) load. We show a simple example of this in Figure 3-13, in which a store can be anywhere prior to the dependent load.

We show a more complex example in Figure 3-14, which is an example memory order that is valid in any acoherent type in which stores affected by a checkin interleave in memory order with loads from another processor. Because the checkout on thread 2, CO_2, occurs before the checkin CI_1, the loads LD_1 and LD_2 are not guaranteed to see stores ST_1 and ST_2. In the order shown, LD_1 sees the new value but LD_2 does not. In general, it is valid for stores preceding a

checkin to interleave with any combination of either loads or stores from different processors, subject to the usual constraints laid out in the model. To achieve the effect the programmer presumably intended in this example, she should use memory transactions (i.e., $\text{Store}_{\text{sync}}$, $\text{Load}_{\text{sync}}$) to ensure that conflicting accesses cannot interleave. Below we show how she could use a spinlock to achieve such an effect.

Similarly, checkout actions do not need to be atomic, and can be implemented lazily as described in Chapter 2.

Checkin transfers can be coalesced. The ASM model does not require stores from different addresses to appear in the same order in both memory order and program order. This has an important implementation implication; structures that buffer stores locally do not need to maintain FIFO order. In particular, this feature allows stores to leak from working memory in any arbitrary order in weak acoherence and allows implementations of checkin to write back stores to repository memory in any arbitrary order.

Checkout is nonblocking. Checkout operations do not have to cause a processor to block. If a processor must stall due to an inter-thread dependence, it will not have to until a load following a checkout, as defined by constraint (3-14). Of course, an implementation must be able to detect the stalling load's causal dependency on a previous store. In Chapter 4 we show one possible mechanism for doing so with low overhead.

3.5.3 Synchronization Examples

We conclude our analysis of the model by running through two examples of synchronization in ASM. First, we show an implementation of a spinlock in which the lock is allocated in a different segment than the data it is protecting. Second, we walk through the design of the THE

Lock Routine

```
01: lock:
02:   LL r1, Lock           ; the load in LL is a Load_sync
03:   BNEZ r1, acquire      ; spin if Lock=0
04:   ADDI r1, r1, 1        ; Lock=1
05:   SC r1, Lock           ; the store in SC is a Store_sync
06:   BEQ r1, acquire       ; check for SC atomicity
07: enter:
```

Unlock Routine

```
08: unlock:
09:   ANDI r1, r1, 0        ; clear the lock value
10:   ST_sync r1, Lock      ; release lock with a normal Store
```

Critical Section

```
11:   JAL lock              ; lock is acquired after returning
12:   CO critical_segment   ; check out the critical segment
13:   LW r0, critical       ; load previous value
14:   …                     ; process…
15:   SW r0, Critical       ; update
16:   CI critical_segment   ; check in the critical segment
17:   JAL unlock            ; release lock
```

Figure 3-15. Simple spinlock implementation in ASM

task queue described in Section 2.5.3.3, using the model to show why checkout and checkin are placed where they are.

Spinlock Example. In this example we show how programs can achieve mutual exclusion using the ASM consistency model. Consider the code in Figure 3-15 that implements a simple spinlock using a MIPS-like ISA. Assume the load in a Load Linked (LL) instruction is a $Load_{sync}$ operation, the store in a Store Conditional (SC) instruction is a $Store_{sync}$ operation, and both the lock and the critical object are allocated in (different) weak acoherent segments. Below we will show that in any execution of the critical section, if a thread acquires the lock, then the load of `critical` on line 13 will observe the last update from the previous store of `critical` on line 15 as expected. We will also address the subtlety of why the unlock store on line 10 must be a

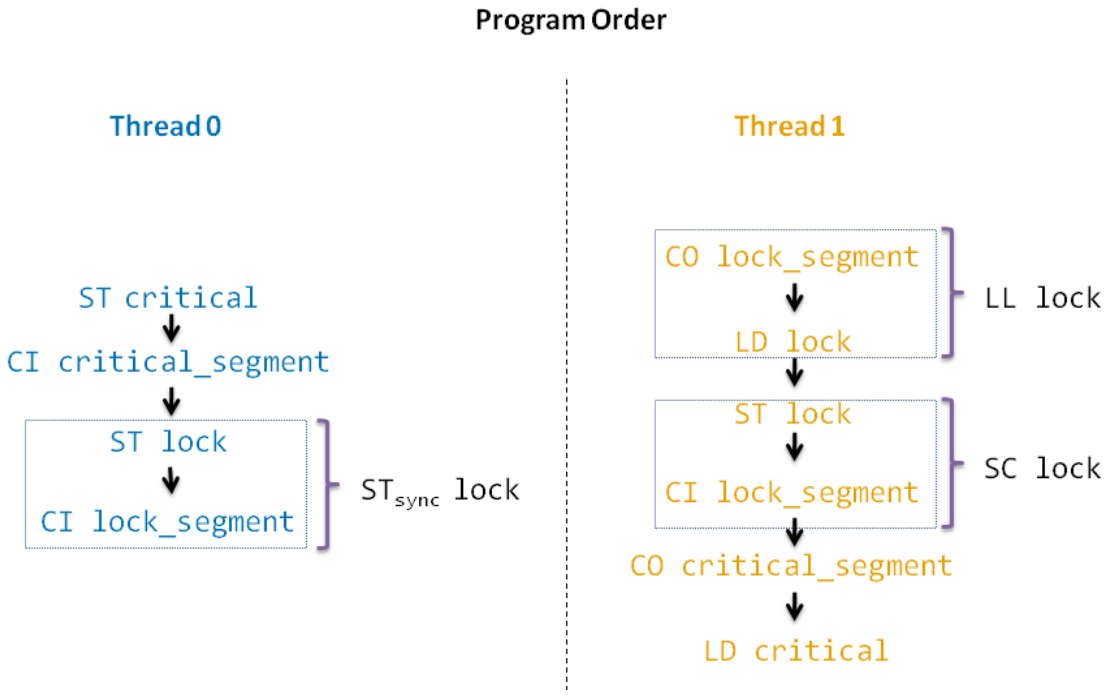

Figure 3-16. Memory operations from critical section in Figure 3-15

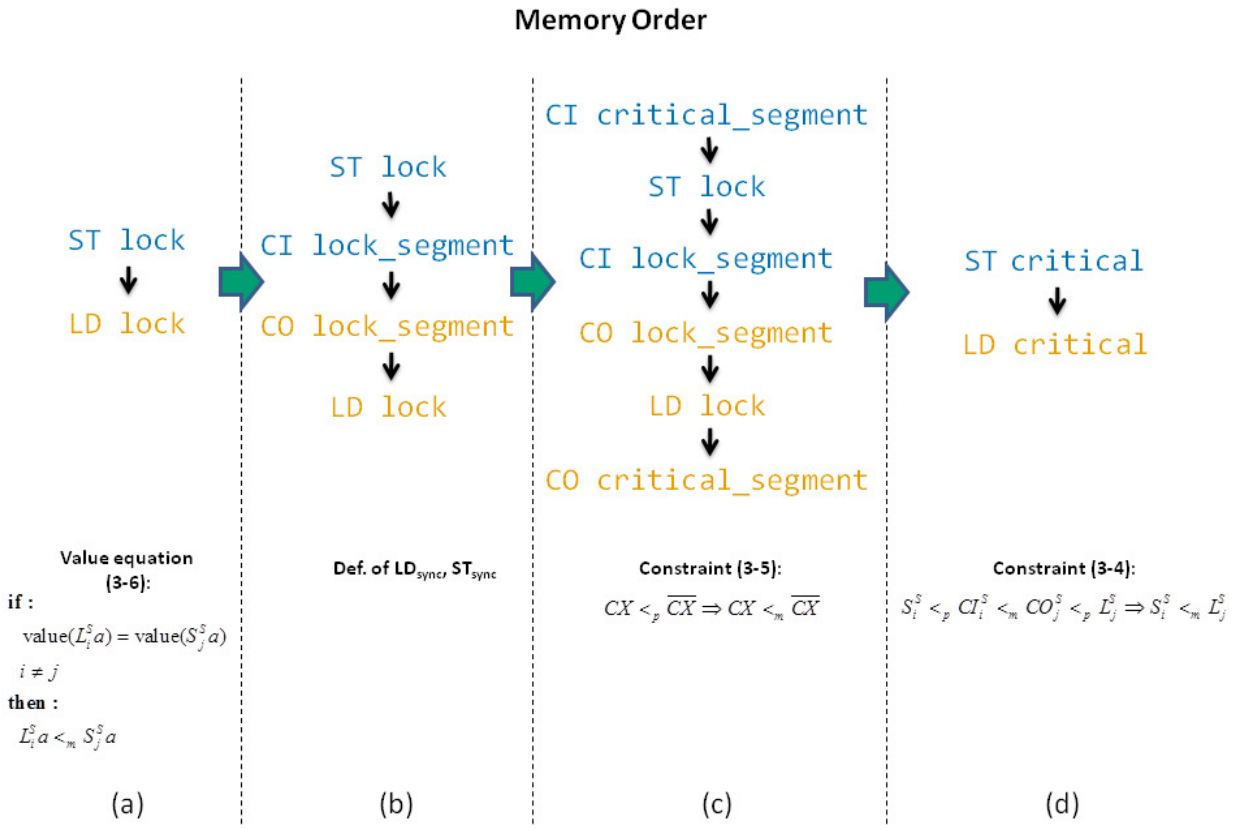

Figure 3-17. Memory orderings that must be true in any execution of Figure 3-16

$Store_{sync}$, which is in contrast to other well-known models like TSO in which unlocks can occur with normal store operations.

In Figure 3-16 we show the relevant memory operations that occur between two threads racing on the critical section. Thread 0 already holds the lock when the sequence begins and Thread 1 acquires the lock after it has been released. In the figure we break memory transactions into their component parts and group them with a dashed box.

In Figure 3-17 we show the progression of rules that prove if Thread 1 acquires the lock from Thread 0, then Thread 1 must see the update to critical made by Thread 0. In (a), we show that because of the value equation, the store that unlocks **`lock`** must come before the load that begins the sequence to lock **`lock`**. Since both of those are memory transactions, we can then include the associated checkin and checkout in memory order, shown in part (b). With a checkin and checkout from each thread placed in memory order, we can include the remaining checkout and checkins with constraint (3-15). Finally, in part (d) we know that by constraint (3-14) the store that updates **`critical`** must come before the load that reads **`critical`** (but we can't say anything about their order relative to the operations in part (c)). Further, by the value equation and part (d), **`LD critical`** must observe the value of **`ST critical`**.

From Figure 3-17 we can see why the store that unlocks **`lock`** must be a $Store_{sync}$. If it were not, there would be no rule to get from (c) to (d), and thus no guarantee that Thread 1 would see the update to **`critical`**. Pragmatically, the $Store_{sync}$ is needed because in ASM there is no guarantee that a store will eventually "leak" from working memory, and so without the checkin it is possible that no other thread would ever observe the unlock. Further, if the checkin were not atomic with the store that performs the unlock, it is possible that another thread would observe the unlock but not the update of **`critical`**.

Owner Worker/Victim	Remote Worker/Thief

```
01: push() {
02:   tmp = T;
03:   checkin(task pool);
04:   T = tmp + 1;
05: }
06:
07: pop() {
08:   T--;
09:   if (H > T) {
10:     T++;
11:     lock(L);
12:     T--;
13:     if (H > T) {
14:       T++;
15:       unlock(L);
16:       return FAIL;
17:     }
18:     unlock(L);
19:   }
20:   return OK;
21: }
22:
```

```
23: steal() {
24:   lock(L);
25:   H++;
26:   if (H > T) {
27:     H--;
28:     unlock(L);
29:     return FAIL;
30:   }
31:   checkout(task pool);
32:   task* cpy = copy(H);
33:   unlock(L);
34:   return OK;
35: }
```

Figure 3-18. THE protocol in ASM.

This is a replication of Figure 2-19 with the garbage collection code omitted.

Note that the code in Figure 3-15 assumes that the lock is located in a weak acoherent segment. If the lock were held in a coherent read-write segment instead (as it would be in our prototype system, Chapter 4), a special store instruction is not needed for the unlock because of the implicit conversion rule in Table 3-5.

THE Synchronization. We now dissect the synchronization for the THE protocol that was first introduced in Section 2.5.3.3 and is replicated in Figure 3-18. In particular, we discuss why the checkin is required on line 03 and why the checkout is required on line 42. Recall that T and H are allocated in a coherent segment and that the task pool (and thus all task objects) is allocated in a weak acoherent segment. We wish to show that if a thief finds a task in a remote queue, all updates made to the stolen task will be visible for the copy on line 32. We concentrate only the push and steal routines since a thief will not try to read any object that has been popped.

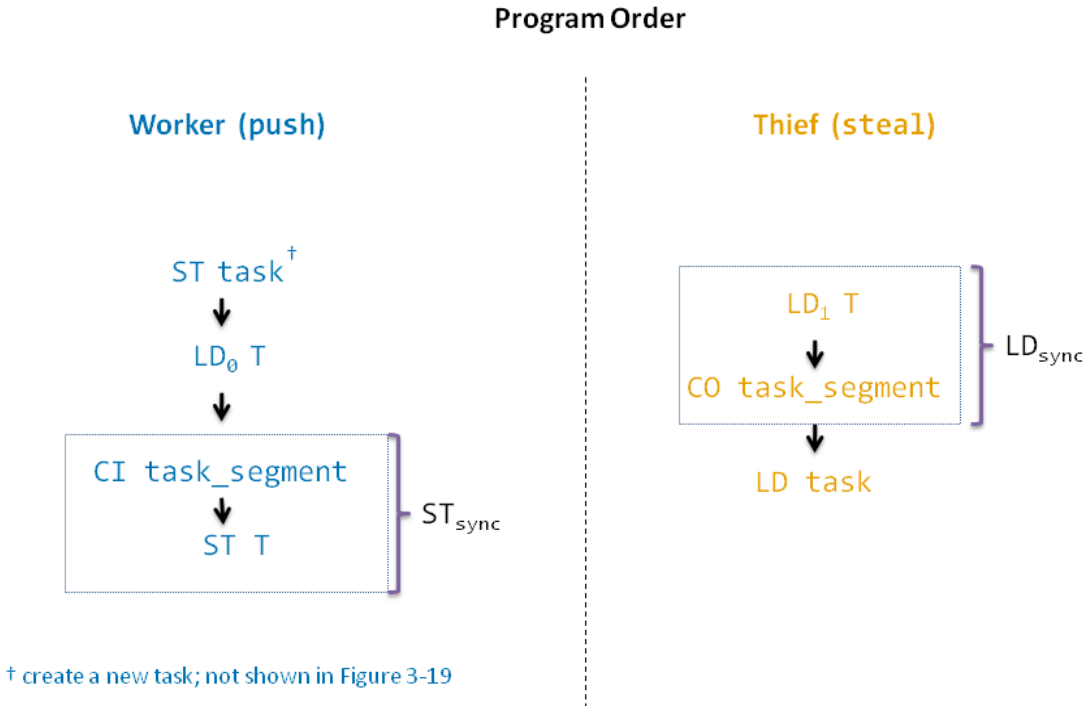

Figure 3-19. Memory operations from the task implementation of Figure 3-18.

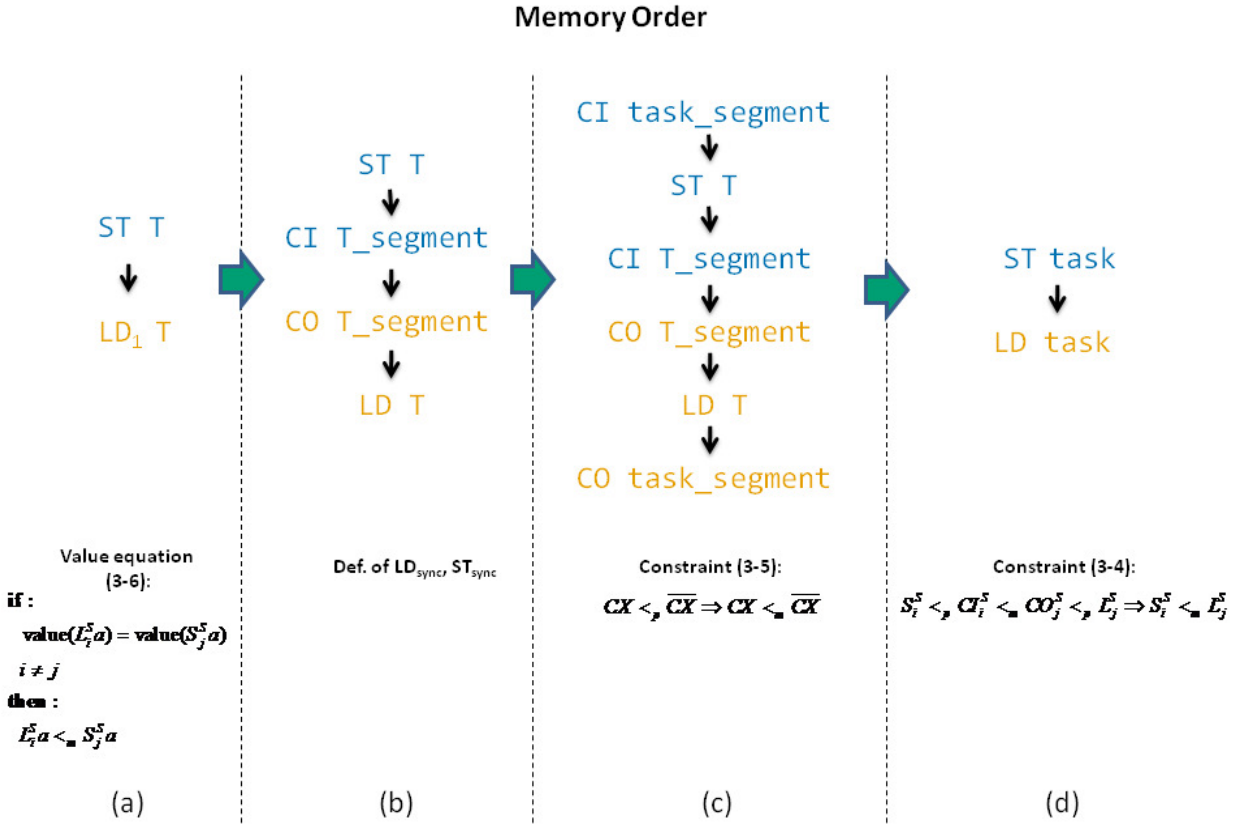

Figure 3-20. Memory orderings that must be true in any execution of Figure 3-19

Also, for simplicity we assume that the value of H does not change over the course of the example.

In Figure 3-19 we show the relevant memory operations that occur if a worker creates a new task, pushes it on its queue, and then a thief steals the task. As we did above, we break the memory transactions into their component parts and outline them with a dashed box (recall that T is in a coherent segment, so any load or store is treated as a $Load_{sync}$ or $Store_{sync}$, respectively). We assume that LD_1 T sees the value produced by ST T.

Figure 3-20 shows the progression of memory ordering constraints that must be true in any execution of the operations in Figure 3-19. First, in (a), LD_1 T must appear in memory order before ST T because LD_1 T observes the value of ST T and they are on different processors. In (b) we include the implicit checkout or checkin that occur atomically with any load or store to a coherent segment. With a checkout and checkin in memory order from both processors, we can include the remaining checkout and checkin in memory order based on constraint (3-15). Finally, from constraint (3-14) we know that **`ST task`** must occur in memory order before **`LD task`**. By the value equation, **`LD task`** will observe the value produced by **`ST task`**, and thus the thief will correctly observe its stolen task.

3.6 Sufficient Conditions for SC Equivalence

While the formal description in Section 3.4 is helpful to hardware designers who must validate that an implementation obeys the memory model, it is arguably too abstract to be useful to application developers who prefer to reason in terms of sequential consistency [88]. For them, we formalize two conditions, *properly-paired* and *lossless*, that, when combined, guarantee that an ASM execution obeys the rules sequential consistency. Programmers are then free to reason

about a program in a sequentially consistent manner as long as they can guarantee that all possible executions of a program satisfy the two conditions. As we show in the next section, this includes all programs that are data-race-free [141].

3.6.1 Two Conditions for SC equivalence

We present two conditions that, if followed, are sufficient to ensure that ASM executions appear sequentially consistent. The first, properly paired, condition ensures that processors do not see others updates until (a) the other processor publishes those updates with a checkin, and (b) the local processor subscribes to those updates with a checkout. The second, lossless, condition ensures that no store value is lost to others because of a clobbering checkout.

$$\forall L_i^S a, S_j^S a : \text{value}(L_i^S a) = \text{value}(S_j^S a), i \neq j \Rightarrow \\ \exists CO_i^S, CI_j^S : S_j^S a <_p CI_j^S <_m CO_i^S <_p L_i^S a \qquad (3\text{-}21)$$

Figure 3-21. Properly paired condition

Properly Paired. An execution is properly paired if and only if all loads receive values from stores that are checked in before the segment containing the load was checked out. More precisely, if a load $L_i^S a$ on thread i receives a value from store $S_j^S a$ on thread j, then there must exist a checkin CI_j^S and checkout CO_i^S such that $S_j^S a <_p CI_j^S <_m CO_i^S <_p L_i^S a$. Using formal notation:

$$\forall S_i^S a, \ \textit{if} \ \exists CO_i^S : S_i^S a <_p CO_i^S \Rightarrow \\ \exists CI_i^S : S_i^S a <_p CI_i^S <_p CO_i^S \qquad (3\text{-}22)$$

Figure 3-22. Lossless condition

Lossless. An execution is lossless if and only if all stores are followed in program order by a checkin before a checkout. Formally, a lossless execution obeys the following conditions:

The lossless condition guarantees that store values will not be “lost” due to checkout operations. A store can be lost to other processors (i.e., can never lend its value to another

processor's load) if it is checked out before is it checked in, as shown in the example in Figure 3-11. The lossless condition precludes this situation by ensuring that stores are surrounded by checkout and checkin operations in program order. Note that the condition precluded by the lossless condition is not only necessary for sequential consistency, it is necessary to avoid undefined behavior in weak acoherent segments (see Table 3-2).

Given the above definitions, we assert the following Theorem that relates properly paired and lossless (PPLL) executions to sequential consistency:

> **Theorem 3.1.** *All properly paired and lossless executions on an ASM machine are sequentially consistent.*

3.6.2 **Proof**

For the proof we use the structured style promoted by Lamport [109]. Specifically, because of the large size of the proof, we use Lamport's compact numbering option. Rather than enumerating steps with numbers that identify all nesting levels, e.g., 3.1.1.1.2, we instead label steps by nesting depth and step within the nesting level, e.g., ⟨5⟩2. This numbering scheme can produce duplicate labels for proof steps but the labels are never ambiguous. To see why, consider that at any nesting level within a proof, a backwards reference can only point to a step in the proof that shares the same assumptions. Thus, even though 3.1.1.1.2 and 3.2.1.1.2 are both labeled as ⟨5⟩2, a reference ⟨4⟩2 from 3.1.1.1.2 can only refer to 3.1.1.2 because that is the only fourth nested, second step in the proof that shares the exact same assumptions. Similarly, a reference ⟨4⟩2 from 3.2.1.1.2 can only point to 3.2.1.2.

Proof Sketch We prove Theorem 3.1 by showing that the value of a load in a properly paired and lossless execution can be described *entirely* in terms of (a) program order and (b) the global total order of checkout and checkin operations. Specifically, we show that the value of a load

from *any* segment type in a properly paired and lossless execution is equal to either (a) the value of the store most recent in program order if the store occurred on the same processor, or (b), if the store occurred on a different processor, the value of the store latest in program order before the checkin latest in memory order that separates the store and the load. Because the rules for program order and checkout/checkin order are the same in both ASM and SC, any properly paired and lossless ASM execution must also be sequentially consistent.

PROVE: **Theorem 3.1**
ASSUME: Properly Paired and Lossless

<1>1 Choose $\tilde{L}_i^S a$, $\tilde{S}_j^S a$ such that:

$$value\left(\tilde{L}_i^S a\right) = value\left(\tilde{S}_j^S a\right)$$

<1>2 ASSUME: $i \neq j$

DEFINE: $\vec{CO}_i^S \triangleq \max_{\underline{p}}\left(CO, \tilde{L}_i^S a\right)$

$$\vec{CI}_j^S \triangleq \max_{\underline{m}}\left(CI, \vec{CO}_i^S\right)$$

PROVE: $\widetilde{S_j^S} a = \max_{\underline{p}}\left(S, \vec{CI}_j^S\right)$

<2>1 $\exists CO_i^S, CI_j^S : \tilde{S}_j^S a <_p CI_j^S <_m CO_i^S <_p \tilde{L}_i^S a$

PROOF: By definition of properly paired (3-21)

<2>2 $\vec{CO}_i^S$ and $\vec{CI}_i^S$ exist and are unique

PROOF: By <2>1 and definition of max

<2>3 $\tilde{S}_j^S a <_p \vec{CI}_j^S$

<2>4 $\neg\exists \overline{S_j^S a} : \tilde{S}_j^S a <_p \overline{S_j^S a} <_p \vec{CI}_j^S$

PROOF: $\overline{S_j^S a}$ is a contradiction

<3>1 $\tilde{S}_j^S a <_m \overline{S_j^S a}$

PROOF: Def. of $\overline{S_j^S a}$ and Condition (3-13)

<3>2 $\overline{S_j^S a} <_m \tilde{L}_i^S a$

PROOF: <2>2, def. of $\overline{S_j^S a}$, $\vec{CI}_j^S$, and $\vec{CO}_i^S$, and condition (3-14)

<3>3 ASSUME: S is *not* a strong/best-effort segment

PROVE: $\overline{\widetilde{S}_j^S a}$ is a contradiction

<4>1 $\text{value}\left(\widetilde{L}_i^S a\right) \neq \text{value}\left(\widetilde{S}_j^S a\right)$

PROOF: <3>1, <3>2, and value equation (3-19)

<4>2 Q.E.D.

PROOF: <4>1

<3>4 ASSUME: S is a strong/best-effort segment

PROVE: $\overline{\widetilde{S}_j^S a}$ is a contradiction

<4>1 $\max_{\underline{p}}\left(CO, \overline{\widetilde{S}_j^S a}\right) <_m \overline{\widetilde{S}_j^S a}$

PROOF: Def. of lossless (3-21) and constraint (3-18)

<4>2 $\text{value}\left(\widetilde{L}_i^S a\right) \neq \text{value}\left(\widetilde{S}_j^S a\right)$

PROOF: <3>1, <3>2, <4>1, and value equation (3-19)

<4>3 Q.E.D.

PROOF: <4>2

<3>5 Q.E.D.

PROOF: <3>3 and <3>4

<2>5 Q.E.D.

PROOF: <2>4

1<3> ASSUME: $i = j$

PROVE: $\widetilde{S}_j^S a = \max_{\underline{p}}\left(S, \widetilde{L}_i^S a\right)$

<2>1 $\neg\exists \overline{\widetilde{S}_i^S a} : \widetilde{S}_j^S a <_p \overline{\widetilde{S}_i^S a} <_p \widetilde{L}_i^S a$

<3>1 $\widetilde{S}_j^S a <_m \overline{\widetilde{S}_j^S a}$

PROOF: Def. of $\overline{\widetilde{S}_j^S a}$ and Condition (3-13)

<3>2 ASSUME: S is *not* a strong/best-effort segment

PROVE: $\overline{\widetilde{S}_j^S a}$ is a contradiction

<4>1 $\widetilde{S}_j^S a <_p \widetilde{L}_i^S a$

PROOF: Def. of $\overline{\widetilde{S}_j^S a}$

<4>2 $\text{value}\left(\widetilde{L}_i^S a\right) \neq \text{value}\left(\widetilde{S}_j^S a\right)$

PROOF: <3>1, <4>1, value equation (3-16)

<4>3 Q.E.D.

PROOF: <4>2

<3>3 ASSUME: S is a strong/best-effort segment

PROVE: $\overline{\widetilde{S}_j^S a}$ is a contradiction

<4>1 ASSUME: $\max_{\underline{p}}\left(CO, \widetilde{L}_i^S a\right) <_p \overline{\widetilde{S}_i^S a}$

PROVE: $\overline{\widetilde{S}_j^S a}$ is a contradiction

<5>1 $\text{value}\left(\tilde{L}_i^S a\right) \neq \text{value}\left(\tilde{S}_j^S a\right)$

PROOF: Assumption <4>1 and def. of $\overline{S_i^S a}$

<5>2 Q.E.D.
PROOF: <5>2

<4>2 ASSUME: $\overline{S_i^S a} <_p \max_{\underline{p}}\left(CO, \tilde{L}_i^S a\right)$

PROVE: $\overline{S_j^S a}$ is a contradiction

<5>1 $\max_{\underline{p}}\left(CO, \overline{S_i^S a}\right) <_m \overline{S_i^S a}$

PROOF: Def. of lossless (3-21), constraints (3-15), and (3-17)

<5>2 $\text{value}\left(\tilde{L}_i^S a\right) \neq \text{value}\left(\tilde{S}_j^S a\right)$

PROOF: <3>1, <5>1, and value equation (3-19)

<5>3 Q.E.D.
PROOF: <5>2

<4>3 Q.E.D.
PROOF: <4>1 and <4>2

<3>4 Q.E.D.
PROOF: <3>2 and <3>3

<2>2 Q.E.D.
PROOF: <2>1

1<4> $\text{value}\left(\tilde{L}_i^S a\right)$ is independent of the position of $\tilde{S}_j^S a$ in memory order.

$\text{value}\left(\tilde{L}_i^S a\right)$ only depends on program order and the relative positions of checkout and checkin operations in memory order.
PROOF: <1>2 and <1>3

1<5> Program order is the same in ASM and SC

1<6> A subset of an ASM execution containing only checkout/checkin operations is an SC execution
PROOF: constraint (3-15)

1<7> Q.E.D.
PROOF: 1<4>, 1<5>, an 1<6>

3.6.3 **Analysis**

Theorem 3.2. *Coherent read-write accesses are sequentially consistent with respect to one another.*

Proof: The subset of execution from coherent operations is always lossless and properly paired. A store is always implicitly followed by a checkin, and so the subset execution is

lossless. A store is always implicitly followed by a checkin and a load is always implicitly preceded by a checkout, so all pairs of stores and loads must be separated by a checkin-checkout pair and consequently the subset of execution must be properly paired. Because properly paired and lossless executions are legal for SC, coherent accesses are sequentially consistent with respect to one another. □

3.7 Sequential Consistency for Data-Race-Free and Lossless Executions

In this subsection we will prove that any ASM execution that is both data-race-free and lossless is also a valid sequentially consistent execution. We define the *happens before relation,* $\underline{hb}$,as the irreflexive transitive closure of program order and the total order of checkin and checkout [3].

Data Race Two accesses form a data race if and only if they are to the same address and are not ordered in $\underline{hb}$.

Data-Race-Free An execution is data race free if there are no data races.

Lemma 3.1. *All data-race-free executions are properly paired.*

Proof: For any store *S* and load *L* to the same address that occur on different processors, if they are ordered in $\underline{hb}$ there must be a checkout CI_j^S and checkout CO_i^S such that (a) CI_j^S appears before CO_i^S in the total order of checkin and checkout and (b) CI_j^S appears after *S* in program order and (c) *L* appears after CO_i^S in program order. Any load that observes the value of a store on a different processor must obey the same conditions, and is thus properly paired. □

> **Theorem 3.3.** *All data-race-free and lossless executions in ASM are sequentially consistent.*

Proof: Lemma 3.1 and Theorem 3.1. □

3.8 Weak = Strong for Data-Race-Free and Lossless

When an execution is data-race-free and lossless, there is no difference between weak and strong acoherence. This observation allows programmers to reason in terms of strong acoherence even if they are running programs on systems that only implement weak acoherence. To prove our claim, we show that all three of the strong acoherence properties are true in weak acoherence when an execution is properly paired and lossless.

> **Theorem 3.4.** *All data-race-free and lossless executions in ASM are sequentially consistent.*

Proof: We prove Theorem 3.4 by showing that the three extra properties of strong acoherence (see Section 3.4.3) are true for weak acoherence by virtue of being properly paired and lossless. By Lemma 3.1, weak acoherence is equivalent to strong acoherence for data-race-free and lossless executions.

1. Stores are not visible remotely until checked in. Proof: Assume there is a store $S_i^S a$ that is observed by a load $L_j^S a$, where $i \neq j$. Because the execution is properly paired, there must exist a checkin CI_i^S and checkout CO_j^S such that $S_i^S a <_p CI_i^S <_m CO_j^S <_p L_j^S a$. Thus, any store that is observed must have been checked in prior to the load that observed it.

2. Stores are not clobbered. Proof: Definition of lossless.

3. All reads are repeatable between checkin/checkout operations. Proof: By the definition of weak acoherence, if there is an unrepeatable read then there must be a remote store between two loads in memory order, i.e., $\exists S_j^S a : L_i^S a <_m S_j^S a <_m \overline{L_i^S a}$. Because the execution is properly

paired, however, there must also exist a checkin CI_j^S and checkout CO_i^S such that $S_j^S a <_p CI_j^S <_m CO_i^S <_p \overline{L_i^S a}$. Because $L_i^S a$ and $\overline{L_i^S a}$ are from the same processor, by transitivity we know that they must be separated by CO_j^S such that $L_i^S a <_p CO_i^S <_p \overline{L_i^S a}$. Thus, $\overline{L_i^S a}$ does not exist. □

3.9 ASM-CVS: An Operational Model of ASM Consistency

The framework used to describe ASM consistency so far, which is adopted from the framework of Weaver and Germond [176], is good both as a foundation to develop formal proofs for comparison to other models and to provide insights for hardware designers. It is less good as a conceptual tool for programmers trying to understand the model, especially for programmers intentionally writing code that is not DRF (e.g., to write a speculation runtime). To help bridge the conceptual gap, we have developed an operational model that easier to grasp than the equations used in this Chapter. Because we have not yet formally proved that the operational model is equivalent to the functional model, we present the operational model in Appendix B.

3.10 Related Work

Checkout and checkin are similar to acquire and release in Release Consistency [70]. Both are used to signal an intent to subscribe to or publish shared data, and under both models a program can have an incorrect execution if the operations are omitted. The ASM model does have several notable differences, however. First, in ASM checkout and checkin apply only to a segment, not the entire address space like acquire and release. Second, in ASM checkout is a clobbering action whereas in RC acquire is not (acquire will not discard a store that was not updated elsewhere).

Third, checkout and checkin are nonblocking operations, whereas acquire and release are. In RC, a thread cannot begin a new instruction until all previous acquires and releases have completed. In contrast, in ASM a thread can continue to execute until a future load or store encounters a consistency dependency. This is similar to the distinction made by Adve and Hill when comparing RC to DRF-1 [4].

DRF-1 is a consistency model that guarantees sequential consistency for data-race-free programs and leaves all other executions undefined. Unlike its predecessor, DRF-0, DRF-1 uses paired synchronization (like acquire/release) when defining data race freedom. ASM is not compatible with DRF-1 because of the effect of clobbering stores. Like DRF-1, though, ASM allows programmers to assume sequential consistency for well formed executions (data-race-free and lossless). Unlike DRF-1, ASM defines some behaviors that are not data-race-free, allowing software to take advantage of the features of ASM (e.g., write a speculation runtime).

TreadMarks is a software DSM system that introduced the notion of Lazy Release Consistency [102], in which update propagations are purposely postponed until the time of an acquire. This is in contrast to most Release Consistent implementations that propagate updates as soon as resource constraints will allow. Similar to LRC, in ASM updates are postponed until a checkin operation. Due to this similarity, ASM builds on the TreadMarks multiple-writer protocol that allows more than one observer to hold a read-write copy of data at the same time. TreadMarks uses a diff mechanism to resolve any conflicts that occur. ASM-CMP, described in the next Chapter, also allows multiple writers by granting access to a cache block to multiple processors at once, and uses the cache bitmask as a diff mechanism.

Like Entry Consistency (EC) defined by the Midway SDSM system [23], ASM allows multiple consistency domains within the same program. EC has semantics similar to Release

Consistency but at a finer granularity. In EC, synchronized data is explicitly paired with its protecting synchronization variable. Release and acquire operations are performed on specific synchronization variables, and only affect the data they protect. In contrast, other weak models guarantee that the entire address space becomes consistent on release and/or acquire operations. ASM is similar to EC in that it allows software to perform consistency operation on a fine granularity (i.e., only on data that matters), but does so using segments rather than synchronization object pairing.

Memory models for high level languages also require explicit program annotations for correctness, for example **`atomic`** in C++ [29] and **`volatile`** in Java [122]. A compiler for these languages could use the information presented by these keywords to determine when it is appropriate to insert checkout and checkin operations.

For a comparison to a variety of models, we reproduce Figure 8 from Adve and Grharachorloo's memory consistency tutorial [2] in Table 3-8 and include the ASM model. Notably, the ASM model does *not* allow a write to be seen earlier on one processor before it can be seen by all. This is an artifact of the consistency model being defined in terms of a single total memory order.

Table 3-8. Comparison to other consistency models.

Relaxation	W->R Order	W->W Order	R->RW order	Read Others' Write Early	Read Own Write Early	Safety Net
SC [108]					✓	
IBM 370 [127]	✓					Serialization instructions
TSO [176]	✓				✓	RMW
PC [70]	✓			✓	✓	RMW
PSO [176]	✓	✓			✓	RMW,STBAR
WO [56]	✓	✓	✓		✓	Synchronization
RCsc [70]	✓	✓	✓		✓	Release, acquire, nsync, RMW
RCpc [70]	✓	✓	✓	✓	✓	Release, acquire, nsync, RMW
Alpha [159]	✓	✓	✓		✓	MB, WMB
RMO [176]	✓	✓	✓		✓	Various MEMBAR's
PowerPC [46]	✓	✓	✓	✓	✓	SYNC
ASM	✓	✓	✓		✓	Checkout, checkin

4

ASM-CMP: An Initial Design

In this chapter we describe the first design of an ASM system called ASM-CMP. ASM-CMP is a single chip multiprocessor system designed deliberately to push the ASM model. It is based on the MIPS ISA and is modified in two major ways. First, ASM-CMP employs a technique called Long Pointer Propagation (LPP) that allows traditional memory segmentation (i.e., base + offset) with a flat, one-dimensional address space. This feature gives software flexibility to use ASM semantic segments in interesting ways without adding undue complexity for programmers. Second, ASM-CMP adds several new instructions in support of the ASM programming model, including the obvious checkout and checkin instructions.

To support fast and efficient ASM operations, ASM-CMP uses a novel cache called a Decoupled Metastate Cache (DMC) that physically separates data from its metastate in the cache layout. By decoupling the metastate, a DMC can complete critical segment-granularity operations (i.e., invalidate all segment data) in a single cycle.

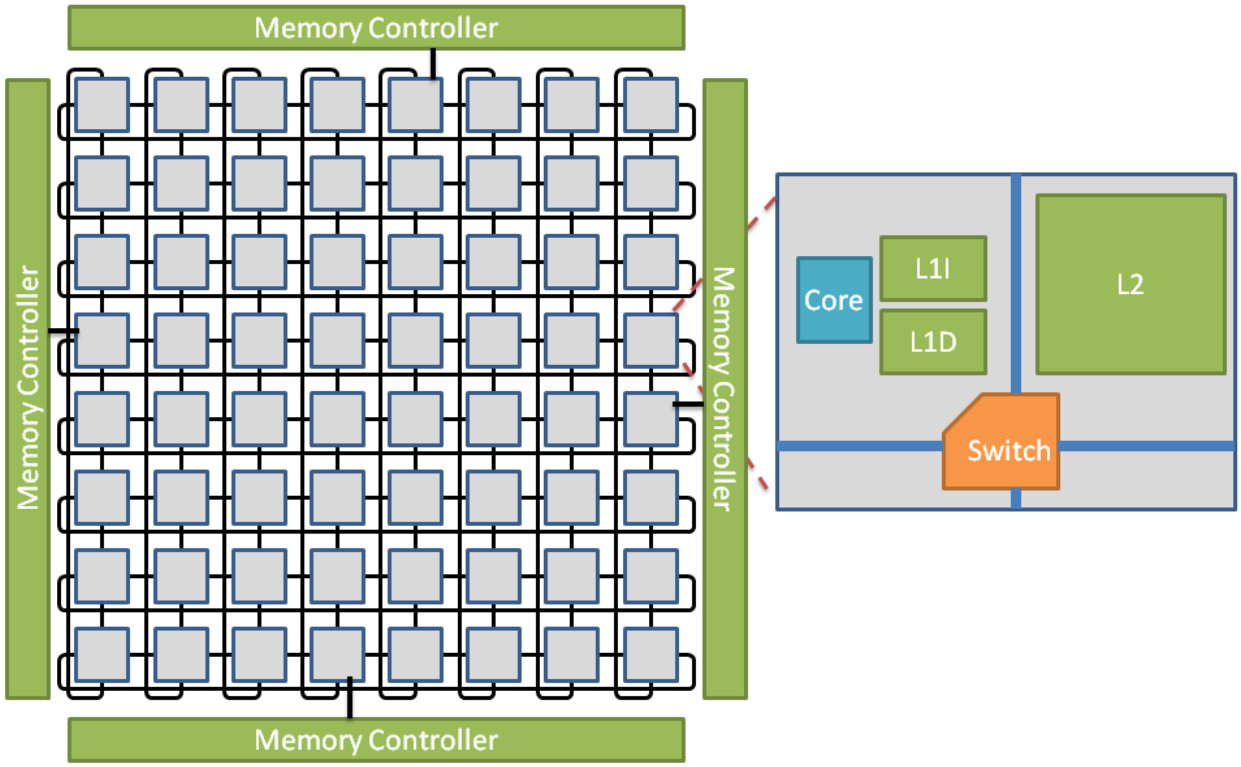

Figure 4-1. A single-chip multiprocessor baseline design for ASM-CMP.

4.1 Overview

ASM-CMP is designed to push the envelope of the ASM model for a single-chip multiprocessor like the one in Figure 4-1. Specifically, we assume that ASM-CMP processors have private, split L1 caches and a shared, banked, L2 cache, and a MIPS ISA. ASM-CMP supports segments as a first class abstraction, avoids a distributed coherence protocol, and uses a special type of cache to support fast and efficient checkout/checkin operations.

ASM-CMP supports segments using a novel architectural technique called Long Pointer Propagation (LPP) that allows software to use segments without having to explicitly identify a segment descriptor on every memory access. With LPP, high level languages (even C) can remain segment agnostic so that existing languages still work on ASM-CMP without needing any semantic or syntactic changes.

To support the acoherent memory model, ASM-CMP exploits the observation that the on-chip cache hierarchy of the baseline in Figure 4-1 closely resembles the acoherent abstraction. In particular, ASM-CMP treats the L1 cache as if it were working memory and the L2 cache (plus main memory) as if it were repository memory. On a checkout, ASM-CMP invalidates all data from a segment that is valid in the local L1 cache (forcing a read from repository memory on the next load). On a checkin, ASM-CMP flushes all dirty data from a segment back to the L2 (allowing the next remote read to see the updated value).

With that model in mind, ASM-CMP employs two mechanisms that streamline checkout and checkin. First, ASM-CMP uses a structure called a Decoupled Metastate Cache (DMC) that tracks the read and write sets of acoherent segments in the L1 cache. The DMC is designed so that a checkout operation can complete in one cycle and supports fast checkins by providing single-cycle access to a summary of the data that must be flushed. Second, ASM-CMP uses a firmware routine to establish a total order of checkout and checkin operations in the system. This firmware routine does not require any new hardware support.

ASM-CMP supports coherent read-write accesses by reading and writing them directly to the shared L2 cache, as is done in recently proposed CMP systems [95]. As we show in Section Chapter 6, because coherent data is accessed infrequently, the longer latency of these accesses do not negatively affect performance. To reduce network traffic and contention when threads spin on a coherent read-write address, ASM-CMP uses a **`monitor/mwait`** mechanism similar to that in x86 [94]. Below we will show that supporting such a mechanism, even in the absence of a coherence protocol, is simple and does not require much state (e.g., around 1K per core).

For both coherent read-only and private types, ASM-CMP accesses the memory hierarchy as if it were totally incoherent. Read and writes both access the L1 cache first, and then proceed to

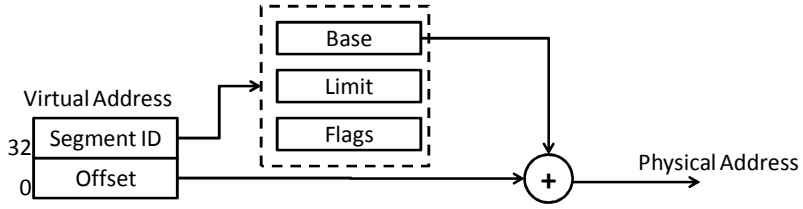

Figure 4-2. Virtual address format and translation

larger levels on the hierarchy on a miss. No attempt is made to ensure consistency among processors for these segment types.

4.2 Segments

ASM-CMP is fundamentally a segmented architecture. All memory references are segment-relative, and physical addresses are calculated by adding an offset to a segment base address. Every segment also has a limit (a.k.a. bound) that is checked on every access. There is no paging support in ASM-CMP, though there is no fundamental reason it could not be added in future iterations. An ASM-CMP virtual address consists of a segment offset and a virtual segment ID, as show in Figure 4-2. On a memory access, the virtual segment ID is decoded (using mechanisms described below) into a physical base address, limit, and segment flags that are used to perform address translation.

In ASM-CMP, segments are contiguous and disjoint (though two descriptors can alias to the *same* segment). It is the responsibility of system software to enforce the disjoint property, and behavior is undefined if that constraint is violated.

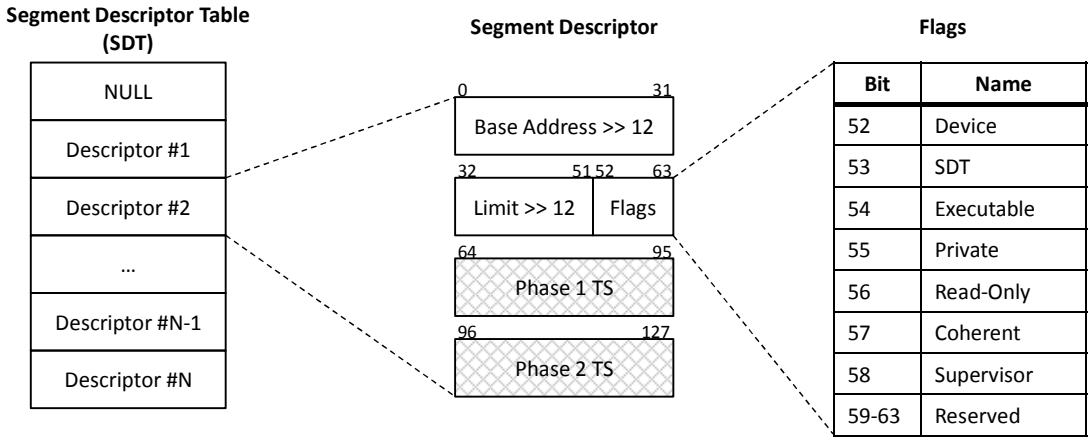

Figure 4-3. Segment Descriptor Table and Segment Descriptor format.
The high 64 bits of the Segment Descriptor (hatched) are opaque to software.

4.2.1 Segment Virtualization

Segments are virtualized so that (1) the operating system has complete control over the physical placement of a segment in memory, (2) a user process can never access its own physical base address, and (3) segments are isolated in a process (unless the operating system chooses to alias a segment among processes). Segment descriptors are held in a protected memory by a collection of tables called the Segment Descriptor Tables (SDTs). To a first order, SDTs are similar to tables in other segmented architectures (e.g., the LDT/GDT in x86 [93]). A single SDT defines an address space, such that one SDT holds all of the segments in a process.

In Figure 4-3 we show the contents of a single 128-bit SDT entry. The first 64 bits describe the static properties of a segment, including the base address, limit, and flags. In ASM-CMP the segment size must be a multiple of 4KB, so twelve low order bits of the base address and limit are omitted. With this format ASM-CMP supports a 44 bit physical address space and a 4GB (2^{32}) maximum segment size. There are twelve flag bits per descriptor, which are used to hold the permissions and semantic type (e.g., acoherent) of the segment. The second 64 bits of the descriptor are opaque to software; more on their function can be found in Section 4.3.2.5.

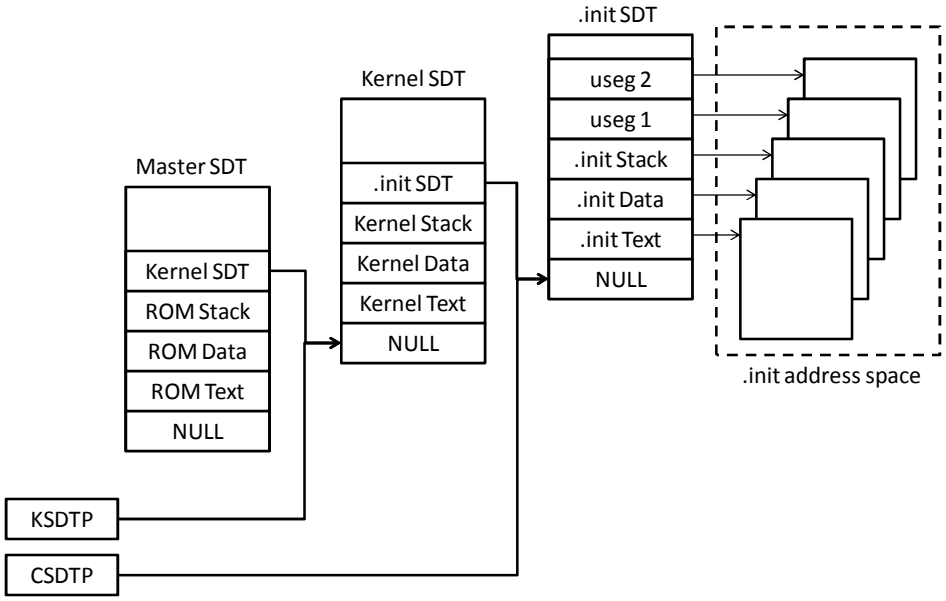

Figure 4-4. The SDT tree for the first (`.init`) process.

SDTs form a tree rooted at a special table called the Master SDT. The Master SDT is created automatically after system reset, and initially contains a single segment descriptor representing the bootup ROM text. The ROM code itself may create a few additional segments for its own execution (e.g., a data and stack segment) by writing new entries in the Master SDT. Before relinquishing control to the kernel, the ROM code is responsible for creating a new child SDT to hold the kernel address space (see Figure 4-4).

There are two protected (i.e., directly accessible only to supervisor software) registers used in conjunction with SDTs. First, the Current SDT Pointer (CSDTP) register holds the physical address of the current SDT for the running process. All memory references (supervisor and kernel) use the CSDTP to perform virtual address translation. In ASM-CMP the virtual segment ID is actually the index of the descriptor in the process' SDT. The translation mechanism is logical only; translations are actually cached, as described below.

The second register is the Kernel SDT Pointer (KSTDP) that by convention always holds the physical address of the kernel's SDT, and is used to speed up context switches. The hardware will never use the KSDTP for address translation, and it is up to software to ensure that the KSDTP is moved into the CSDTP at the appropriate time. We provide the KSDTP register because early on in the context switch, kernel memory is not easily accessible (a kernel segment descriptor is not loaded in any other register).

Privileged software creates a new segment by writing the base, limit, and flags of an available SDT entry. Hardware does not treat SDT addresses specially; the translation mechanism ensures that once an entry is created, all future loads and stores to the address will access the correct segment.

Two entries (in addition to the NULL entry) in every SDT have a special meaning. The first entry in an SDT always holds the descriptor for the program's text segment and the second always holds the descriptor for the program's static data segment. All instruction fetches implicitly reference the first segment descriptor. By convention, register **`$zero`** always holds the segment descriptor for the text segment and register **`$gp`** holds the segment descriptor for the data segment.

In ASM-CMP, SDTs are allocated in coherent memory, so that they are never written into a private L1 cache. If software changes the attributes of a segment, the operating system can ensure those changes are globally visible (i.e., "segment shootdown") by (a) updating the SDT entry and (b) tainting the register file of other processors. We describe how the tainting mechanism in Section 4.2.2.2, but for now the reader may assume it is a sufficient way to force processors to refresh their cached segment translations.

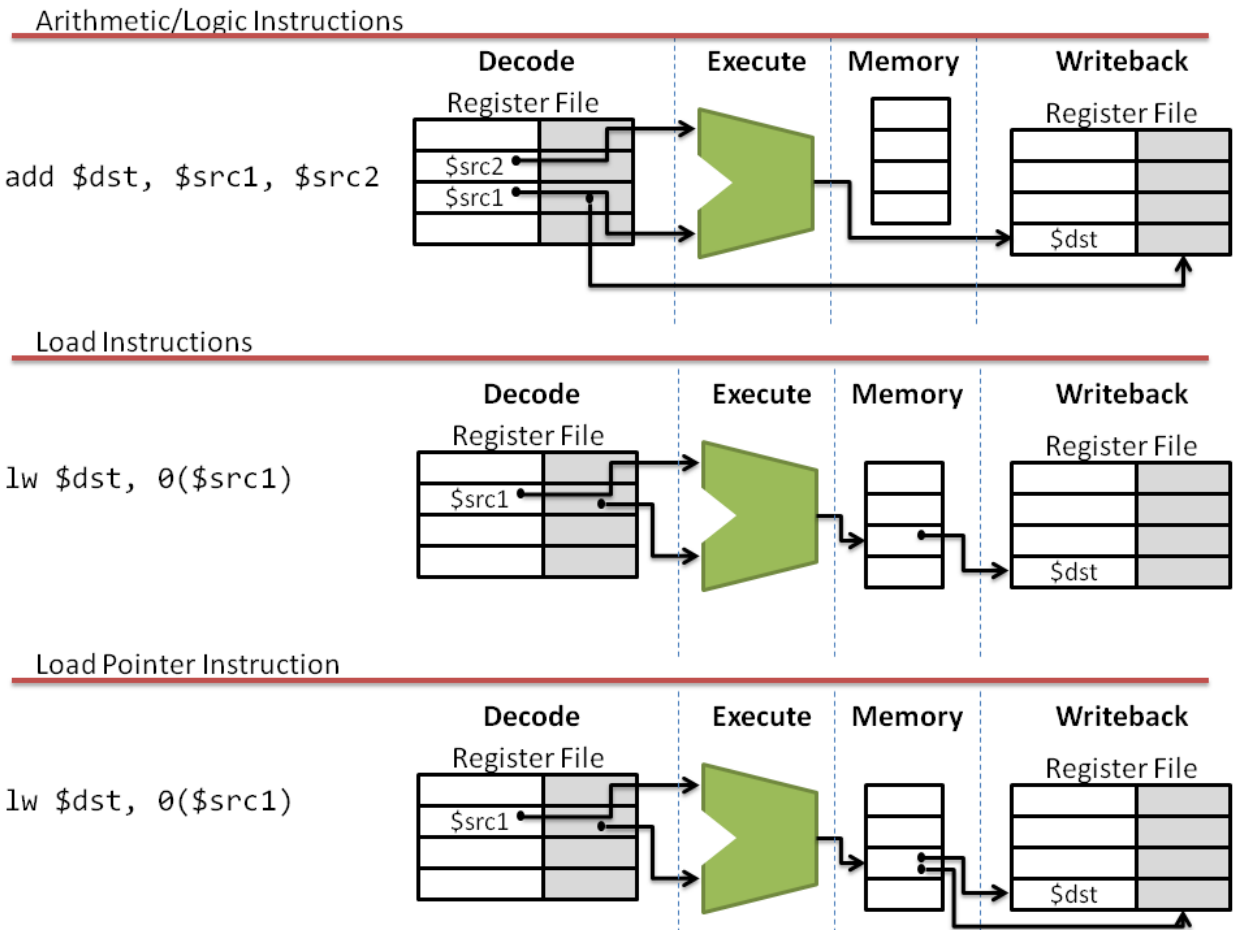

Figure 4-5. Three LPP examples.

We show the pipeline diagram for three types of instructions. In the diagram, the shaded boxes are register sidecars. On top is an example arithmetic instruction, which is modified in ASM-CMP so that the sidecar of the first source operand propagates to the sidecar of the destination. In the middle is a load word instruction, which in ASM-CMP uses the base address contained in the sidecar to translate a virtual address. Finally, on the bottom is a load pointer instruction that decodes the segment ID portion of a pointer variable in memory and stores it in the destination register sidecar.

4.2.2 Long Pointer Propagation

ASM-CMP employs a novel architectural technique called Long Pointer Propagation (LPP) that provides segmentation without the need to explicitly reference a segment during a memory access. With LPP, software, especially high level languages (even C), can remain segment-agnostic and does not require any language modifications.

LPP works by propagating a segment descriptor along the dataflow of all pointer variables. General purpose registers are augmented with a hidden segment sidecar that holds a decoded segment descriptor. Any time a pointer is dereferenced (which in MIPS/ASM-CMP will always involve a register source operand), the associated descriptor is used to perform address translation. The segment sidecar propagates from the (first) source operand of any arithmetic/logic instruction to the sidecar of the destination register, as shown in Figure 4-5.

When a variable must spill from a register, ASM-CMP uses two special instructions that maintain pointer dataflow propagation. The Store Pointer (**`sp`**) instruction writes the offset and segment ID of the pointer in the source operand to the location specified by the destination register. Load Pointer (**`lp`**) performs the inverse operation, as shown in Figure 4-5.

In Figure 4-6 we show how a simple **`memcpy`** function works with LPP. In the example, a static read-only string is copied into a dynamic global buffer. The first thing to note is the initialization of both the **`g_dst`** and **`g_src`** pointers on lines 25-30. Both are 64 bits long; the first 32 bits hold the offset and the second 32 bits hold the segment ID. Both happen to point to offset zero within their respective segments. In the case of **`g_dst`**, the segment ID is set to two because by convention the global data segment is always the second segment ID (see above). On the other hand, **`g_src`** points to the segment with ID one, which again by convention always points to the combined text/read-only data segment.

On lines 34 and 35 the main function loads both **`g_dst`** and **`g_src`** into registers using the lp instruction. At that time, the virtual segment IDs are decoded and placed in the respective register sidecars. When those registers are later dereferenced on lines 43 and 44, the decoded segment descriptor is used for address translation. Finally, lines 46 and 47 show how the pointers

C Code

```
01: char buf[6];
02: char* g_dst = buf;
03: char* g_src = “Hello”;
04:
05: int main(…) {
06:   memcpy(g_dst,
07:          g_src,
08:          16);
09:   return 0;
10: }
11:
12: int memcpy(char* dst,
13:            char* src,
14:            int size) {
15:   int i;
16:   for (i=0;i<size;i++)
17:     dst[i] = src[i];
18:   return 0;
19: }
```

Assembly Code

```
20: .rodata
21:   .asciiz “Hello”
22: .data
23: buf:
24:   .zero 6
25: g_dst:
26:   .word 0    ; segment offset
27:   .word 2    ; segment ID for globals
28: g_src:
29:   .word 0    ; segment offset in .rodata
30:   .word 1    ; segment ID for text/rodata
31:
32: .text
33: main:
34:   lp    $a0, 0($gp) ; $a0 = g_dst
35:   lp    $a1, 8($gp) ; $a1 = g_src
36:   li    $a2, 16
37:   jal   memcpy
38:   move  $v0, $0
39:   jr    $ra
40:
41: memcpy:
42:   blez  $a2, exit_memcpy
43:   lb    $t0, 0($a1) ; $t0 = mem[$a1+$a1.sc.base]
44:   sb    $t0, 0($a0) ; $t1 = mem[$a0+$a0.sc.base]
45:   subi  $a2, $a2, 1
46:   addiu $a0, $a0, 1 ; $a0=$a0+1, $a0.sc=$a0.sc
47:   addiu $a1, $a1, 1 ; $a1=$a1+1, $a1.sc=$a1.sc
48:   b     memcpy
49: exit_memcpy:
50:   move  $v0, $0
51:   jr    $ra
```

Figure 4-6. Example of memcpy function in ASM-CMP assembly

propagate along the dataflow of arithmetic instructions. In the example the segments “propagate” to the same register, but the principle would still apply if the destination register were different.

Notably, the **`memcpy`** function on lines 12-19 will work for *any two segments*, including a copy within the same segment. Thus, programmers using a high level language can create code without having to think about segments other than at allocation time (e.g., lines 01-03).

4.2.2.1 Segment Lookup Tables

Referencing the actual SDTs on every segment ID decode could quickly become a performance bottleneck. To alleviate that pressure, ASM-CMP uses a small (e.g., eight entries) associative table that caches recently referenced segment descriptors. The segment lookup table is indexed by virtual ID, and as such is also tagged with an address space identifier.

The segment lookup table is similar to a TLB in that is an associative cache for address translation, but differs in several key ways. First, it is smaller than a TLB (e.g., 8 vs. 64 entries) because, by their nature, there are far fewer segments than pages in a program. Second, there is only one segment lookup table versus two (instruction and data) TLBs. ASM-CMP does not need a lookup table for instruction accesses because the CSDTP always holds the decoded descriptor for the current text segment. Third, the segment lookup table is accessed less frequently than a TLB. The table is accessed on all load pointer instructions, and is *not* accessed on a store pointer, normal load, or normal store.

4.2.2.2 Register Tainting

Periodically the decoded descriptors in the register sidecars can become invalid (e.g., on a context switch). To make sure software does not access an invalid descriptor, the sidecars can be tainted, which forces the system to re-decode the segment on the next attempted dereference. Sidecars are tainted when one of two events occur. First, they are tainted when CSDTP is written (i.e., a context switch). Second, they are tainted on a checkout. We discuss why they must be tainted on a checkout in Section 4.3.

4.2.3 **Compiler Support**

The ASM-CMP segment mechanism requires special support from a compiler, though the changes are localized, generally at code generation time, and do not require structural

modification of the compiler infrastructure. A compiler must support three tasks. First, it must be able to distinguish pointer types at code generation time so that it can emit load pointer/store pointer instructions instead of the standard load word/store word. This information is usually present for other (optimization) reasons at code generation time, and so is generally a small imposition. A compiler may also chose to prohibit typecasts from integers to pointers, which in ASM-CMP would generally have undefined behavior on a dereference.

Second, a compiler must be able to ensure that for any instruction involving pointer arithmetic, the operand holding the pointer is always in the first source operand. Again, this is easily accomplished if the compiler tracks pointer types up to the code generation phase. If the arithmetic involves two pointer sources, the compiler can choose randomly which one to place first under the assumption that both pointers are in the same segment. Performing arithmetic on pointers from different segments would be considered bad programming practice by many because they are by definition unrelated to each other. In our experience thus far, we have never seen such a scenario.

Third, the compiler needs to insert segment IDs for statically initialized data, as shown in lines 27 and 30 in Figure 4-6. The compiler already has the information needed to insert the correct IDs because it is already capable of defining the binary segments (e.g., .text, .rodata, .data, …) that correspond to the correct segment IDs.

4.3 Acoherence Support

Below we describe the ASM-CMP acoherence implementation from two viewpoints. First, we provide a theoretical foundation by describing ASM-CMP operationally and showing why it is a

valid implementation of the ASM consistency model. Then we will provide a practical overview, giving specific details of the hardware structures and mechanisms.

4.3.1 **Theory**

Operationally, ASM-CMP obeys the ASM consistency model for acoherent segments by following four basic policies:

1. A checkout invalidates all segment data in the private L1 cache.
2. A checkin flushes all dirty segment data in the private L1 cache back to the shared L2.
3. A checkout or checkin completes when:
    a. Its actions (policy 1 or 2) are complete.
    b. Globally, all previous checkins to the same segment complete.
    c. Locally, all previous checkouts and checkins in program order complete.
4. A load following a checkout stalls until that checkout completes.

Together, these policies work together to ensure that any checked-in store will be visible to any subsequently checked-out load. The policies make checkout and checkin appear atomic when in reality they are both non-blocking and non-atomic operations in ASM-CMP. The first and second policies combine to ensure that the first load following a checkout will read from the same logical memory (i.e., the L2 cache) that the last store before the previous checkin was written. Policies three and four make sure that the load does not try to read the L2 early, i.e., before all previous checkins have completed.[8]

In order to implement policy three, specifically policy 3b, ASM-CMP must have a way to determine the global total order of checkout and checkin operations to the same segment. For this ASM-CMP uses logical timestamps. The timestamps are assigned with the following rules:

T1. A checkout *CO* gets a timestamp *CO.TS* equal to the timestamp of the previous (in real time) checkin to the same segment.

[8] This policy is actually conservative. A load actually only needs to wait until all checked-in stores to the load's location have completed, not all checked-in stores in general.

T2. A checkin *CI* gets a timestamp *CI.TS* equal to the timestamp of the previous (in real time) checkin to the same segment plus one.

With timestamps assigned, a partial order of checkouts and checkins to the same segment can be established with the following rule:

T3. A checkin or checkout from segment S CX^S happened before another checkin/checkout $CX^{S`}$ on the same segment if $CX^S.TS < CX^S.TS$.

Rule T3 results in a partial order because it does not distinguish between concurrent checkouts that may receive the same timestamp (rule T1). However, the policies described above do not require such a distinction.

4.3.1.3 ***ASM-CMP is a Valid Implementation of ASM***

We now seek to show that the policies for acoherent management outlined above result in a valid implementation of the ASM consistency model. Readers already convinced may choose to skip this section. We focus on the conditions related to acoherent operation, specifically memory ordering constraints (3-14) and (3-15) and the value equation (3-16).

To facilitate the discussion, we first make an observation. While the memory order defined by the consistency model is meant to be a purely hypothetical and unobservable order, in ASM-CMP it actually has a real counterpart. In ASM-CMP, memory order is the order in which loads, stores, checkouts, and checkins arrive at the shared L2 cache. Stores that do not access the L2 (hit in the L1) appear in memory order with the *next* store to the same address that accesses the L2. Loads that do not access the L2 (hit in the L1) appear in memory order with the *previous* load to the same address that accessed the L2. Checkouts and checkins appear in memory order in the order that their timestamp loads read from the L2.

First we show that ASM-CMP respects memory ordering constraint (3-14). Consider a load L, store S, checkout CO, and checkin CI such that $L <_p CO$, $CI <_p S$, and $CI <_m CO$ (i.e., CI

has a timestamp less than CO). To respect the constraint, ASM-CMP must ensure that L appears in memory order (accesses the L2) after S. This will always be the case. CI will not complete until the store has been written to the L2 (rules 2 and 3). CO will not complete until CI completes (rule 3). Finally, L will not issue until CO completes. By transitivity, L will access the L2 after S.

Second, we show that ASM-CMP constructs a total order of *all* checkouts and checkins that respects program order, as required by constraint (3-15). From rules T1-T3 we are left with disjoint partial orders of the checkout and checkin operations from each processor. We can join them into a unified partial order with the addition of a fourth rule:

> T4. Any checkin or checkout *CX* from any segment happened before another operation *CX'* if *CX* appears in program order before *CX'*.

Combined, rules T1-T4 form a global partial order that respects local program order. The partial order can be turned into a total order by assigning priorities to processors, as is commonly done with Lamport Scalar Clocks [107]. Because rules T3 and T4 are logical rules not actually constructed at runtime by ASM-CMP, we have shown that the timestamps assigned by rules T1 and T2 are sufficient for obeying memory constraint (3-15).

Finally, we show that the load value equation is respected by ASM-CMP. The load value equation guarantees that a load will receive the value of a store either (a) most recent (with respect to the load) in memory order or (b) latest in memory order from a store previous in program order. To show this we use the L2 memory order analogy. If a load receives the value of a store from another processor, it will get the value returned by the L2, which will be the last store seen from that location. If the load bypasses from a local store, it will receive the value of a store that will appear in memory order after the load when that store is either evicted or flushed.

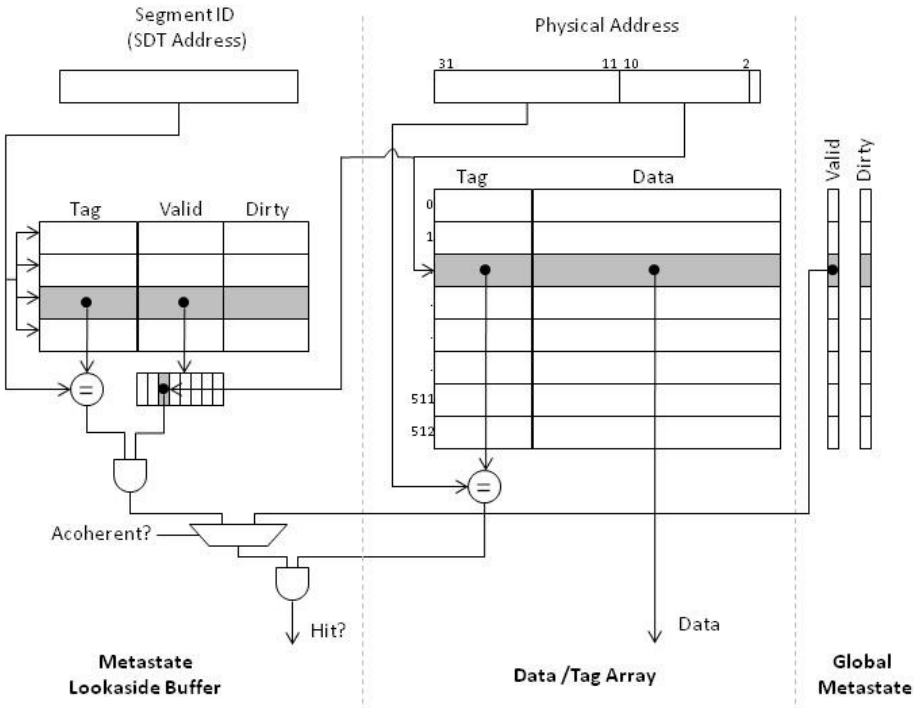

Figure 4-7. Example read from a direct-mapped DMC

4.3.2 Implementation

In ASM-CMP, the mechanisms needed for acoherence support are orchestrated by a per-processor finite state machine called the Acoherence Engine. The engines interact with special L1 caches called Decoupled Metastate Caches (DMCs) to efficiently implement the checkout and checkin actions required by policies 1 and 2 above. The engines also coordinate with each other through memory to establish the checkout and checkin timestamps required by policy 3. Below, we will discuss the design of a DMC, show the specific timestamp algorithm used, and then put it all together to show how the Acoherence Engine implements both checkout and checkin.

4.3.2.4 ***Decoupled Metastate Cache***

All L1 data caches in ASM-CMP are organized as DMCs. A DMC differs from a normal cache in two ways. First, the usual block metadata (i.e., valid/dirty bits) are physically decoupled

from the data blocks and are stored in aggregate form in global metastate registers, shown in Figure 4-7. Second, a DMC also tracks the *acoherent* dirty and valid blocks in the cache on a per-segment basis in a structure called the Metastate Lookaside Buffer (MLB). Separate per-segment metadata is not tracked for non-acoherent data.

When the DMC accesses or modifies any block in the cache, the decoupled bits in the aggregate global metastate registers are managed just as they are in a normal cache, e.g. by setting the dirty bit on a write. A block in the DMC is only valid if the corresponding bit in the global metastate register is set. If a block happens to be in an acoherent segment, the DMC will also make the *same* corresponding updates/accesses to the dirty and valid bits in that segment's entry in the MLB, as shown in Figure 4-7. MLB entries are identical in structure to the global metastate registers (i.e., have as many bits as there are blocks in the cache). Because a block can only belong to one segment, the same bit cannot be set in multiple MLB entries.

The MLB is a small (e.g., 8 entries) associative cache that is tagged by segment ID. The MLB is accessed in parallel with the data array on any acoherent read or write (and not on coherent or private access). When a block is evicted from a DMC, both the segment and global valid/dirty bits are updated. Evictions access the MLB differently than demand accesses because at eviction time there is no segment ID. Rather, the entry in the MLB is identified by doing to the lookup on the block's valid bit, which can only be set by one entry.

If the MLB runs out room and must evict an entry, all of the dirty data from the evicted segment is also written back and all valid lines are invalidated. In our workloads, this never happens for an MLB with eight entries.

A DMC can quickly perform two notable segment-related operations. First, a DMC can bulk invalidate an entire segment by XORing the segment valid bits in an MLB entry with the global

valid bits held in the global metastate register. Second, the DMC can provide, in a single cycle, the *exact* locations of all dirty blocks in a segment by reading the dirty register in an MLB entry. Below we will show how the Acoherence Engine can use these operations to perform fast checkouts and checkins.

4.3.2.5 ***Timestamps***

In ASM-CMP, timestamps perform two functions. First, they establish the order that checkout and checkin operations issue. Second, they are used to detect when previous operations are complete. For these tasks, ASM-CMP uses two timestamps per segment, both of which are stored in the opaque bits of the segment's descriptor entry (see Figure 4-3).

The first timestamp, TS_{issue}, always holds the timestamp of the most recent checkin that issued. When a checkout begins, it is assigned a timestamp equal to TS_{issue}. When a checkin begins, it is assigned a timestamp equal to TS_{issue} plus one. Also, when a checkin receives a timestamp, TS_{issue} is incremented. This algorithm follows the timestamp rules laid out in Section 4.3.1.

The second timestamp, $TS_{complete}$, always holds the timestamp of the most recent checkin that completed. Before checkout completes, it waits for $TS_{complete}$ to become greater than or equal to the timestamp the checkout received when it issued. Before a checkin completes, it waits for $TS_{complete}$ to become equal to the timestamp the checkin received when it issued minus one. Upon completion, the checkin updates $TS_{complete}$ to equal the checkin timestamp.

Both timestamps are managed with existing mechanisms for managing data (including monitor/mwait functionality described in Section 4.4). Timestamps do not require any new hardware support.

4.3.2.6 Acoherence Engine

The Acoherence Engines (AEs) are finite state machines that orchestrate checkout and checkin. An AE accepts incoming requests from a processor and responds with an appropriate message. Like a cache controller, AEs can be (and indeed are in ASM-CMP) non-blocking, and can handle multiple outstanding requests by using a Miss Status Holding Register (MSHR) [105]. In the Acoherence Engine, MSHRs are allocated and deallocated in FIFO order, as described in more detail below.

An AE accepts three types of requests: **`checkout`**, **`checkin`**, and **`query`**. The first two initiate acoherence operations as expected. A **`query`** request responds with a message that indicates whether or not a particular segment (identified with the request) has an outstanding incomplete **`checkout`** or **`checkin`**. **`query`** requests enable non-blocking checkout and checkins by giving processors a mechanism to detect ordering dependencies.

In particular, the pipeline issues **`query`** requests in conjunction with a Segment Lookup Table access. If the **`query`** response indicates that a checkout from that segment *is* incomplete then the pipeline will stall the load. Because the pipeline also taints the sidecar registers on checkout operations, all loads following a checkout must perform a segment lookup and therefore must also perform a **`query`**, ensuring that dependencies will be detected.

The AE responds to both checkout and checkin requests immediately without waiting for them to complete unless there are no more MSHRs, in which case the requests are blocked.

Upon receiving a **`checkout`** request, the Acoherence Engine performs two operations in parallel. First, it orders the checkout operation using the algorithm described above. Second, the engine sends an invalidate segment request to the DMC and removes the corresponding entry in the MLB. When the timestamp algorithm is complete, the segment is invalidated, and the

checkout is at the head of the MSHR queue, the checkout is complete and the MSHR is deallocated.

Upon receiving a checkin request, the Acoherence Engine performs two operations in parallel. First, it orders the checkin operation using the algorithm described above. Second, the engine reads the segment's dirty metastate from the MLB and then proceeds to issue a writeback to the DMC for each bit that is set. Note that, unlike the checkout operation, the MLB entry is not removed because after a checkin the data is still valid. When both operations are complete and the checkin is at the head of the MSHR queue, the checkin is done and the MSHR is deallocated.

4.3.2.7 ***Multiple Writer Support***

In ASM-CMP it is possible for multiple cores to obtain read-write copies of a block at the same time. Without special attention, ASM-CMP might lose updates if two cores concurrently modify different bytes within the same cache block. To prevent lost updates, ASM-CMP augments cache lines in the L1 data cache with a dirty byte bitmask that indicates which bytes within a block have been modified. When the block is written back, the bitmask accompanies the data and only the dirty bytes are written to the shared L2 cache. More compact representations are likely possible, but we leave that to future work.

4.3.2.8 ***Best-effort Support***

ASM-CMP provides best-effort acoherence support by performing a checkout operation before a block from a best-effort segment can be evicted from the L1 cache. Additionally, ASM-CMP will raise an exception that allows the software to deal with the best-effort failure.

4.4 Coherence Support

In ASM-CMP, coherent data is treated differently depending on whether it is read-only or read-write. Read-only coherent data is effectively treated as incoherent data in the memory hierarchy. All accesses read/write from the L1 cache and misses propagate to larger levels of the hierarchy without any attempt to keep data consistent.

Coherent read-write data skips the L1 altogether and enters the memory hierarchy at the L2 cache. Because the L2 is globally shared, this ensures that coherent updates are seen immediately by all processors. While there is a performance penalty for the long latency access to the L2, our results in Chapter 6 show that the overall impact is negligible. There are at least three reasons for this. First, the L2 access on a coherent read-write access is faster than a traditional L2 hit because the block does not wait for an L1 miss before sending the request to the L2. Second, contended blocks (e.g., those holding contended locks) actually perform better when written to the shared L2 rather than ping-ponging between private caches as they would in a conventional invalidation coherence protocol. Third, ASM-CMP introduces **`monitor/mwait`** instructions to eliminate the negative effects of spin locking on coherent data in the L2.

The **`monitor/mwait`** instructions allow a processor to idle while it waits for a location to change. The monitor instruction arms an address, and the mwait instruction blocks (puts the processor in an idle state) until the monitor is triggered. The instructions have best-effort semantics; on a return from **`mwait`**, the location is not guaranteed to have changed (**`nop`** is a valid implementation).

The astute reader may notice that the **`monitor/mwait`** function is similar to the function of a conventional cache coherence protocol (specifically an update protocol). There are several

reasons why the **`monitor/mwait`** functions are simpler, though. First, the mechanism can be incorrect. The **`mwait`** instruction does not guarantee that a location has changed, only that it may have. Second, at any given time there can only be two addresses per core that are being monitored; one from normal software stream and one from the acoherence engine waiting for a segment timestamp to change. Thus, on average, it suffices to monitor just two locations per cache bank.

With these observations in mind, ASM-CMP implements **`monitor/mwait`** by adding two registers per cache bank that hold the block address being monitored and the ID of the core that is waiting. Every incoming writeback checks the registers for a match. It there is a hit, a wakeup is sent and the register is freed. If there are already two active monitors at a bank and a third arrives, the third is immediately signaled with a wakeup, effectively degrading into a polling spin at the requesting processor.

Recall from Chapter 3 that coherent loads and stores are considered synchronization operations with implicit checkouts/checkins. For that reason, a coherent load or store will stall until all previous checkins in program order have been *ordered* (not necessarily completed). This is a subtle rule. As long as the checkin has been ordered (i.e., has received a timestamp), then any subsequent checkout that is causally dependent on the coherent store will enforce any load delays through the combination of rules 3 and 4 in Section 4.3.1.

4.5 Private Support

Like coherent read-only data, private data is effectively treated as incoherent cached data. Unlike coherent read-only data, private data is managed with an exclusive caching policy. If a block is cached it will reside in either an L2 bank or exactly one private L1, but not both. This

Table 4-1. Instruction modifications and additions in ASM-CMP compared to MIPS

***The first category only shows examples representing an entire class of modified instructions.**

Name	Format	Description
Modified Instructions*		
Add	`add $dst, $src1, $src2`	`$dst = $src1 + $src2` `$dst.sc = $src1.sc`
Store Word	`sw $src, C($dst)`	`mem[$dst + C + $dst.sc.base] = $src`
Load Word	`lw $dst, C($src)`	`$dst = mem[$src + C + $src.sc.base]`
Segment Instructions		
Load Segment	`ls $dst, $id, $sdtp`	`$dst = 0` `$dst.sc = xlate($seg_id, $sdtp)`
Store Pointer	`sp $src, C($dst)`	`mem[$dst+C] = $src` `mem[$dst+C+4] = $src.sc.idx`
Load Pointer	`lp $dst, C($src)`	`$dst = mem[$src + C + $src.sc.base]` `$dst.sc=xlate(mem[$src + C + $src.sc.base], CSDTP)`
Acoherence Instructions		
Checkout	`co $src`	`checkout($src.sc)`
Checkin	`ci $src`	`checkin($src.sc)`
Coherence Instructions		
Monitor	`monitor $src`	`monitor_address[proc] = $src + $src.sc.base`
Mwait	`mwait`	`wait for monitor_address[proc] to change`

policy frees up room in the L2 cache by ensuring that blocks guaranteed not to be needed are not wasting space.

Private data *can* be checked in, which may occur on a thread migration. To support checkin if a private segment, the processor's entire L1 cache is flushed of all dirty data.

4.6 New Instruction Summary

We summarize the modified and additional instructions that ASM-CMP uses compared to MIPS in Table 4-1. In the first group, we show only a few select instructions that represent an entire class (e.g., **`add`** represents all arithmetic/logic instructions). There is one instruction in the

Table 4-2. Hardware state used by ASM-CMP.

Component	Size in bytes (per-core)
Metastate Lookaside Buffer Dirty Byte Bitmask Sidecar Registers Segment Lookup Table	512 1024 128 32
`monitor`/`mwait` registers	16
Total	**1704 bytes**

table, Load Segment (**`ls`**), that has not been mentioned yet. **`ls`** is a privileged instruction, and is the only instruction that does not use the CSDTP for address translation. Instead, it uses the second source operand as an SDT pointer, which allows an operating system to access memory outside its own address space without having to load CSDTP.

4.7 Hardware State Summary

In Table 4-2 we summarize the hardware that is added in ASM-CMP compared to a conventional incoherent baseline. We provide state estimates assuming a system with a 32KB, 4-way set-associative L1 data cache.

4.8 Costs of ASM-CMP

While we believe ASM-CMP to be a reasonable design point that can simplify hardware, there are some costs. In this subsection we discuss how the changes to software and the system as a whole that should be considered carefully before choosing to use a system like ASM-CMP.

First, and perhaps most obvious, is that not all existing application software is compatible with ASM-CMP. Some of the changes arise from segmentation (e.g., casting from a pointer to an int and then back to a pointer is not allowed), though most changes are due to acoherence support. Software must be cognizant of communication points and must ensure that an (implicit

or explicit) checkin/checkout call is executed.[9] For performance reasons, software may also need to perform profiling and tuning to optimize segment/checkin/checkout. In particular, software will need to balance the competing effects of frequent checkin/checkout, which make the operations more targeted, with sparse checkin/checkout, which avoid the overheads of the operations (e.g., timestamp acquisition). We note that if the ASM-CMP design were adopted by a less conventional component than a CPU (e.g., a GPU), many of the compatibility issues would be moot.

ASM-CMP may complicate system software. First, because ASM-CMP does not support paging, system software will need to deal with memory fragmentation using other methods. For example, the operating system could place constraints on the maximum size of a segment so that one process does not consume the majority of physical memory. The segmentation may also put pressure on applications to only allocate memory that they actually use. We discuss the fragmentation issue more in Section 4.9.2.

Context switching can also be more complicated in ASM-CMP, as previously discussed in Section 2.3. In particular, the operating system may have to perform cache flushes when a thread migrates between processors.

Supporting coherent I/O (e.g., DMA) in ASM-CMP is not as straightforward as it is in a system implementing a coherence protocol. Presumably, a system would either (a) need to invalidate the I/O region out of the L2 before initiating a transfer, or (b) write the I/O directly into the L2 cache.

[9] Arguably, forcing programs to identify communication is a *feature* that will lead to more efficient programs, not an issue that will unnecessarily complicate programs.

4.9 Related Work

Some embedded systems like the TI C64x [165] provide an internal DMA engine for doing bulk transfers between intra-core caches. This mechanism is similar to how ASM-CMP implements checkin. ASM-CMP only transfers data that has been modified within a segment, though, opposed to the related work that transfers all data within a region regardless of whether or not it has been touched.

Tarjan and Skadron have proposed an imprecise directory for Graphics Processing Units (GPUs) [164] that, like ASM-CMP, is designed to reduce the cost of coherence. Their design, called Sharing Tracker, exploits the fact that the GPU memory model does not require the strict semantics of coherence (in fact, GPUs have no well defined model at all). In most cases, finding *any* version of a block, rather than the latest version, is sufficient. Sharing Tracker uses this observation to reduce the size of a conventional directory by only recording a single location where a block is known to exist. ASM-CMP, in conjunction with the well-defined ASM model, takes this a step further by eliminating a directory altogether. ASM-CMP also provides a mechanism (checkin and checkout) to resolve concurrent versions when they occur.

4.9.1 Cache Coherence Protocols

The Acoherence Engine is similar to a cache coherence protocol in that both are finite state machines responsible for managing shared caches in a multiprocessor. However, an Acoherence Engine is different in ways that arguably make it a simpler design. First, in an Acoherence Engine, requests are generated from one source (the processor) whereas in a coherence protocol requests may come from one of many sources (processors and other coherence controllers). Because all events are local without the possibility of external interference, the number of

distributed race conditions, and thus the number of transient FSM states, is drastically reduced. In particular, the only races in an Acoherence Engine are the timestamp updates, and those are handled with simple atomic operations on memory locations via mechanisms that already exist. Second, the Acoherence Engine does not need to track state for every cached block as most coherence protocols do. As a result, an Acoherence Engine requires less state than a coherence protocol (e.g., does not need a directory).

There are many pioneering examples of systems that, like ASM-CMP, try to reduce coherence state in large multiprocessor designs. Three seminal designs include the Stanford DASH [116] and SGI Origin 2000 [112] and the Scalable Coherent Interface (SCI) [80,187]. The Stanford DASH research machine showed that directory protocols were practical in large systems. They addressed the issue of directory area by tracking sharers hierarchically. The SGI Origin 2000 by Laudon and Lenoski improved upon the DASH by dynamically switching between full bit vector and coarse bit vector representations, enabling a system that could scale to over 1024 processors. The SCI specification differs from the previous two in that it uses an entirely distributed directory and pointer-based sharers tracking. The design trades off area scalability for an increased response latency stemming from potentially long latency list traversals. SCI also introduced the idea of request combining that can reduce memory traffic by coalescing requests to the block that occur close together in time.

Recently there has been a resurgence of research that attempts to tackle the problem of coherence overheads specifically in manycore CMPs. The Tagless Directory proposal by Zebchuk et al. replaces a sparse directory structure with per-set bloom filters [183,185]. They take advtantage of the fact that protocols using sparse directories with inexact tracking are already able to handle false positives in set membership operations. They show that their design

can save up around 50% of area, power, and energy in on-chip directories. ASM-CMP takes this idea several steps further. First, ASM-CMP does not use an on-chip directory at all. Second, the tagless directory (and sparse directories with inexact sharing) work because the cache coherence protocol is augmented to handle false positives, thereby *increasing* the protocol complexity. Instead, ASM-CMP reduces the complexity of conventional coherence.

The Cuckoo Directory proposal by Ferdman et al. [62] replaces a traditional directory (either sparse or duplicate tag) with a directory that uses Cuckoo hashing [146]. The Cuckoo Directory reduces energy requirements by eliminating costly associative lookups and wasteful invalidations. ASM-CMP is able to reduce state further by eliminating a directory altogether.

Fensch and Cintra proposed a coherent scheme for tiled architectures that relies on OS support for correctness [60]. In their system, block permissions are handled exclusively by OS controlled page permissions. Effectively, the OS pins a page to a particular L1 cache by handing out read-write permission to one thread at a time. Their proposal eliminates directories like ASM-CMP, but incurs a significant performance loss (27% on average) that ASM-CMP avoids.

Multiple-writer coherence protocols such as the TreadMarks Tmk protocol [101] and the Princeton HLRC [182] allow multiple read-write copies of a block to coexist in the system. ASM-CMP also allows multiple writers to coexist, but implements resolution between the multiple writers at a much finer granularity. Both Tmk and HLRC piggyback on paged virtual memory to detect when multiple writers exist, and then use a technique called twinning and diffing to resolve differences. In short, the protocols make copies of any pages that might contain concurrent conflicting writers and then generate diffs at synchronization points to determine which locations changed. They use the diffs to ensure that false conflicts arising from the page-level tracking do not subsume true updates. ASM-CMP uses a very similar mechanism,

but does so at the level of a cache block rather than a virtual memory page. ASM-CMP dynamically tracks the byte-level diff of each block using the write bitmask in the L1 caches. When a block is written back to the L2, the bitmask is used to ensure that clean bytes do not overwrite any concurrent updates.

4.9.2 **Segmentation**

While the ASM-CMP segment architecture may seem radical compared to modern paged systems, there is mounting evidence that a return to segments, at least in part, may lie ahead. The singularity project by Larus et al. is a ground-up redesign of an operating system for datacenters [110,111]. They argue in favor of forgoing paging altogether because most datacenter workloads do not require it. There is also evidence from the mobile space that indicates a design without paging may be feasible in that ecosystem. In particular, the iOS operating system by Apple does not implement demand paging [118]. If an application requests more memory than is available, the operating system simply kills another, lower priority, running application. Because iOS only uses paging for protection and fragmentation, it seems plausible that a segment-based design may work well in that market. Finally, the supercomputing community has identified paged virtual memory as one of the top challenges to overcome in order to meet the energy envelope needed for exascale computing [22,104].

The LPP mechanism used by ASM-CMP is similar to guarded pointers proposed by Carter et al. [35]. Like LPP, guarded pointers store a segment descriptor in the upper bits of a pointer variable. Unlike LPP, guarded pointers are completely opaque to software. To read the offset value of a guarded pointer, software must use special instructions that extract the offset and return it in a normal register. All pointer operations, including pointer arithmetic, also require special instructions. LPP, on the other hand, only requires special pointer instructions when are

loaded to/from a register, and permits the use of stock arithmetic/logic instructions for pointer manipulation.

5

Methods

This chapter explores the methods used to evaluate proposals in this dissertation. First, in Section 5.1 we discuss the baseline system that we use as a control in all experiments. Then we provide details of the experimental infrastructure used to collect results. In particular, we outline three components of the infrastructure. In Section 5.1.1 we discuss the design of the *enhanced user mode* simulator that models the hardware systems and that is used to collect all runtime statistics. Here we also discuss the models we use to estimate both the area and energy results. In Section 5.2.4 we provide details of the system software stack that runs inside the simulator, and then in Section 5.2.5 discuss the toolchain used to create the target software. Finally, in Section 5.3, we discuss our workload selection and provide data showing their runtime characteristics. We break workloads into two classes. First, in 5.3.1, we discuss workloads based on the SPLASH-2 suite that are written for coherent shared memory and are minimally modified for ASM. The second class, in Section 5.3.2, contains workloads created specifically for ASM and exploit the full features of the ASM model.

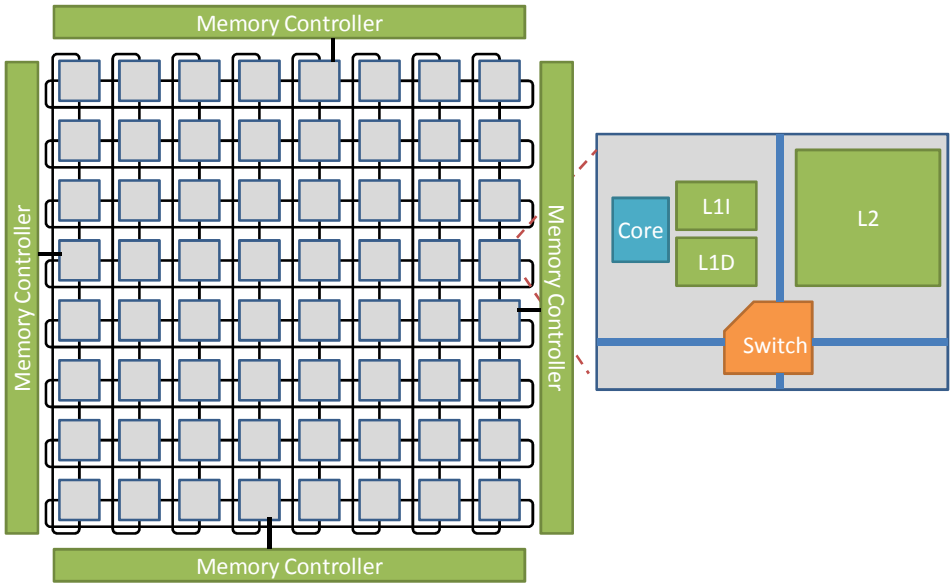

Figure 5-1. Baseline system configuration

5.1 Baseline System Configuration

From a myriad of possibilities, we choose a baseline system that we believe represents one plausible design point for the type of systems ASM targets. The system, shown in Figure 5-1, is similar to previously proposed tiled architectures in both the literature and industry [6,152,171,172,175]. Such a design is becoming increasingly common in multicore ASIC designs because, among other reasons, the regularity can simplify design effort [6,172]. The regularity might also make the integration of heterogeneous components easier by allowing them to reuse existing infrastructure on a tile.

Table 5-1 lists the parameters of this baseline that remain constant across all evaluated systems. The baseline consists of 64 cores arranged on an 8x8 grid connected via a 2D folded torus interconnect (folded connections not shown in Figure 5-1 for simplicity) [49]. Each core

contains a small six-stage, in-order pipeline corresponding to the stages in Figure 5-2. Closely integrated with the core is a 32KB, 4-way set associative L1 instruction cache and a 32KB, 4-way set associative L1 data cache. In addition to a core, each tile contains one slice (128KB) of a unified 8MB 16-way L2 cache and a pipelined network switch. The on-chip network uses static dimension-ordered routing. Each link is 16-bits wide. Observed access times for hits in each level of the memory hierarchy are given in Table 5-2.

Table 5-1. Common system configuration parameters

Component	Configuration
Cores	
Pipeline	in-order, 6-stage (see Figure 5-2)
ISA	RISC (based on MIPS-32 [131,132,133])
Memory	
Memory Controller	Idealized, constant 200-cycle
L1 I-Cache	32KB, 4-way, 64B block, 1 cycle access
L1 D-Cache	32KB, 4-way, 64B block, 1 cycle access
L2 Cache	8MB total, 16-way, 64B block, 3 cycle access
Network	
Switch	4 ports, static dimension-order routing, 5 stage pipe
Link	16-bit full duplex, 3mm, 1 cycle latency
System	
Clock Frequency	2 GHz
Feature Size	32nm

Table 5-2. Calculated access times, excluding any effects from sharing.

Access Type	Range	Average
L1 Hit	1	1
L1 Miss + L2 Hit	12-28	20
L1 Miss + L2 Miss	220-248	234

Fetch	Decode	Execute	Access 1	Access 2	Writeback

Figure 5-2. Pipeline stages.

5.1.1 Baseline Software

Our baseline system runs a UNIX-like environment. We have developed an operating system, detailed in Section 5.2.4, that provides a standard POSIX interface to user applications. The OS also provides some ASM-specific services such as those dealing with segment management.

All software, both system and user, is compiled with a C language toolchain. We do not have support for a dynamic linker, so all applications are statically linked for ASM-CMP.

5.1.2 Conventional Target

We evaluate the baseline system using two different coherence protocols. Both treat the shared L2 cache as a static NUCA design. The first is an inclusive protocol using the standard MESI coherence states [160]. The MESI protocol maintains an on-chip directory implemented as full bit vector co-located with the L2 tags. The second protocol is a non-inclusive MOESI protocol. The MOESI protocol uses a sparse directory at each L2 bank. The protocol, taken from the GEMS infrastructure, assumes that there is a perfect (infinite size) directory at each bank. In our area overhead estimates, we assume that a 512-entry directory is sufficient. This makes our performance evaluation of the MOESI protocol conservative, as it does not handle directory misses.

While the MOESI protocol is higher performing in general than MESI, mainly due to its non-inclusive design, it is also significantly more complex and resource consuming. Table 5-3 lists the properties of each protocol, from which one can gain an understanding of the relative complexities of each. In the table, we calculate the directory state overhead by taking the number of extra bits required for the directory and dividing by the number of bits needed for a directory-less cache of similar size. A more robust empirical evaluation of the area, power, and performance overheads can be found in Chapter 6.

Table 5-3. Baseline coherence protocol properties.

	MESI	MOESI
Stable L1 States	4	7
Transient L1 States	6	8
Stable L2 States	3	13
Transient L2 States	14	46
Total States	27	54
Directory State Overhead	11.9%	20.2%

5.2 Approximation Methods

We evaluate both the baseline system and its ASM variants using an *enhanced user mode* simulator. The enhanced user mode method of simulation is a compromise between two well known simulation techniques: full system and user mode. A full-system simulator models all devices and low-level services in cycle-accurate detail, giving high timing fidelity but also high simulator design overheads and high wall-clock simulation runtimes. User-mode simulators, on the other hand, do not model any devices, and instead functionally emulate system calls using a constant (usually unity) time. User-mode simulators have less fidelity but are easier to build and take less time to run. User-mode simulators can give reasonable timing estimations for workloads that spend the great majority of their execution in user mode, but can give misleading results for those that rely heavily on system services like virtual memory or filesystem operations.

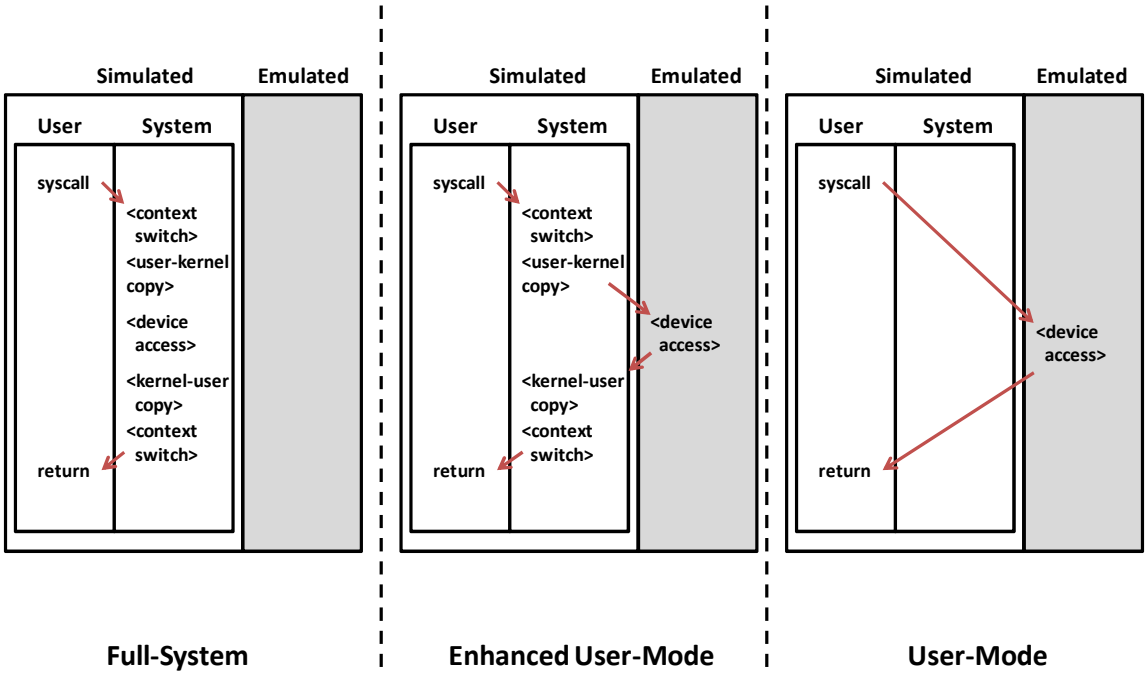

Figure 5-3. An example device access using three simulation methodologies.

The enhanced user mode simulator models parts of the full system stack that are critical to workload performance and functionally emulate other parts that are unimportant to workload performance. Figure 5-3 shows how a typical device access is modeled in the three different simulation methodologies. The enhanced user mode simulator accurately models the interactions between system and user code but emulates the actual device access. As the figure shows, an enhanced user mode simulator requires an operating system kernel that is aware it is running on a simulator, similar to a kernel running on a paravirtualized VMM [15]. Notably, the enhanced user-mode simulator accurately models virtual memory.

In the rest of this section we cover the three main components of the ASM-CMP infrastructure. First, we discuss the design of the ASM-CMP hardware simulator and show what modeling detail it provides for various devices. Second, we discuss ASMIX, the paravirtualized operating system that was designed specifically to run in the simulator. Finally, we will discuss

Table 5-4. Host system parameters

Compiler	gcc 4.1.2 (Red Hat 4.1.2-51)
Linker	GNU ld 2.17.50.0.6-14.el5
Operating System Distribution	Red Hat Enterprise Linux 5
Linux Kernel Version	2.6.18-274.12.1.el5
Ruby (Language [63]) Version	1.8.5
SWIG	1.3.31

the software toolchain used to build both ASMIX and the user applications used in our evaluation, focusing on its attributes and limitations.

5.2.1 Hardware Architecture Simulator

The ASM-CMP simulator is a single-threaded discrete event simulator written in the C language. It was written, run and tested in the environment described in Table 5-4. We defer a description of the low level details of the code to READMEs and comments in the source files, and focus the rest of this subsection on the high level concepts that affect how results should be interpreted.

The ASM-CMP simulator randomizes execution to mitigate the effects of nondeterministic code paths in multithreaded applications, in which small timing perturbations can cause drastically different executions [10]. The randomization arises from the simulator event queue. When multiple events of identical priority are ready at the head of the queue, the ASM simulator will randomly choose among them which event to execute next. Such a situation can happen, for example, when there are two cores ready to fetch a new instruction in the same cycle. When the randomized execution is combined with multiple runs, we gain statistical confidence in the accuracy of the simulator results. We present all simulated results in this document as 95% confidence intervals over ten randomized runs.

We did not validate the simulator against any real hardware system, largely due to the fact that no ASM hardware exists. However, we provide two anecdotes that lend some evidence that

the results are accurate. First, because our simulator manages guest data without the use of an "oracle" functional simulator (e.g., as in timing-first simulators [125,177]), we are confident that the simulations are functionally correct. Because the timing models are closely integrated with the functional models, i.e., it is an execute-in-execute simulation [13,24,25], we can be assured that at the very least we are not modeling impossible execution paths. Second, as described in more detail below, we do validate portions of the ASM simulator to the existing Ruby memory module from gem5 [24]. Ruby has been previously validated against real coherent hardware, giving us an added level of confidence in our results.

5.2.1.1 ***Device Timing Models***

We extract timing data from the simulator by building cycle-accurate models of components in the system. We describe each of those models below.

Cores The core pipeline is modeled after the stages in Figure 5-2. The cores model full register bypassing between stages. The fetch stage supplies instructions without incurring a stall as long as the next PC is in the same cache block as the previous fetch. The execute stage may stall if a register operand is not ready yet or if the register file needs untainting (see details of ASM-CMP segment operation, Chapter 4). We assume there are enough execution units to preclude any resource stalls, and model the floating point unit as a single cycle resource. The access stages stall on cache misses or memory ordering stalls (Chapter 4).

Caches We actually model several different cache organizations depending on the characteristics of the system under test. When modeling coherent caches for baseline measurements, we borrow timing models from the gem5-Ruby memory module [24]. We defer discussion of those models for now and instead focus on the cache models used to simulate an ASM-CMP memory system.

ASM-CMP caches are nonblocking, and model a fixed number (e.g., four) of Miss Status Holding Registers (MSHRs) [105]. Replacements are issued in parallel with demand requests when needed. The controllers work in tandem with the network to implement flow control by refusing to accept new requests when a controller is busy. The L2 caches access tag and data arrays in serial, and may only access the tag array in some situations. The simulator also models the read/write sets in the controllers of private caches.

Memory Controllers We model memory as a fixed latency access. The controllers in the system are placed in the network, though, so on-chip routing delays are modeled.

Switches We model network switches as a simple pipelined router. To model the switch pipeline, we insert a fixed delay equal to the pipeline depth minus one when a new message enters the network and then charge one cycle for actually traversing a switch. The switch models account for bandwidth constraints by delaying messages if the bandwidth limit has already been reached in a given cycle.

Links Network links in the system are modeled as a fixed-latency connection between network components with a bandwidth of one flit per cycle.

Coherence To evaluate ASM-CMP in comparison to conventional shared-memory CMP systems, we have also integrated the gem5-Ruby module into the simulator [24]. Ruby provides a cycle-accurate simulation of a memory hierarchy by modeling caches, coherence protocols, interconnection networks, and DRAM. To integrate Ruby, it is wrapped in an interface identical to that used for the ASM CPU read/write ports and is then used as a drop-in replacement for the ASM memory and network. For all comparison experiments, Ruby and the ASM memory system are configured identically according to the parameters in Table 5-1.

We verified that the coherence and ASM configurations are comparable by running a microbenchmark in each memory system that can report the latencies of different access types (the same microbenchmark was used to collect the information in Table 5-2). The program calculates the latency of a cold miss to DRAM from all cores to all memory controllers, the latency of an L2 hit from all cores to all L2 banks, and the latency of a local L1 hit. The microbenchmark reports the same latencies listed in Table 5-2 for the ASM, MESI, and MOESI configurations.

When integrated with the gem5-Ruby module, guest software in the system will see a sequentially consistent view of memory. While the ASM-CMP memory system presents a weaker model to software, we believe that a performance comparison between the two is valid because *neither* system implements the performance optimizations typically associated with weak consistency models. For example, even though the ASM-1 model permits Load -> Load reordering that would allow a high-performance core to perform non-blocking read operations [2,160] the simple, in-order core in ASM-CMP still implements blocking reads.

5.2.2 **Energy Modeling**

We calculate energy consumption using methodologies previously established in tools such as Wattch [30], and McPat [119]. Specifically, we first calculate the energy consumed by each discrete component in the memory system then report total energy as the sum of all component energies, i.e., for all components in the set C:

$$E(system, execution) = \sum_{i \in C} E(C_i, execution) \qquad (5\text{-}1)$$

We do not calculate the energy of either the processor cores or memory controller/DRAMs. As such, we only report the energy of the on-chip memory system in our evaluation. We assume that the on-chip memory system energy is a significant portion of total system energy, and is thus

Table 5-5. Common CACTI Parameters

Parameter	Value
Technology	32nm
Array Cell Type	L2 cache: itrs-lop, All others: itrs-hp
Array Peripheral Type	L2 cache: itrs-lop, All others: itrs-hp
Operating Temperature	350K
Optimize	NONE
Wire Signaling	Default
Wire Inside MAT	global
Wire Outside MAT	global

an interesting target for energy reduction. Furthermore, since we expect (and results confirm) that ASM will actually reduce the number of DRAM accesses, our relative energy comparison to a coherent system is conservative.

We calculate the execution energies of individual components using two different tools from the literature. For caches, directories, TLBs, and SLTs (see Chapter 4), we use CACTI 6.5 [138,139]. CACTI takes as an input high level characteristics of a cache, such as size and associativity, runs an optimizer to find the best hardware organization, then reports the timing, energy, and latency characteristics. Table 5-5 lists the common parameters we pass to CACTI for all components. Once we acquire the per-access energy from CACTI, we multiply that by the number of accesses over an execution to arrive at the total component energy for a workload. Formally, we calculate energy from a CACTI component as:

$$E(C_{CACTI}, execution) = (E_{ACCESS} * num\_access) + (P_{Leakage} * simulated\_time) \quad (5\text{-}2)$$

Table 5-6. Non-default Orion parameters used by all routers and links.

Parameter	Value
PARM_TECH_POINT	32nm
PARM_TRANSISTOR_TYPE	NVT
PARM_Vdd	0.9
PARM_Freq	2 GHz
PARM_in_port	4
PARM_cache_in_port	2
PARM_out_port	4
PARM_cache_out_port	2
PARM_flit_width	16
PARM_pipeline_stages	5

For switches and links we use Orion 2.0 [173]. Unlike CACTI, Orion reports component power for a given average load rather than a per-access energy estimate. Thus, for switches and links we use simulator statistics to get the average device load during an execution and then run Orion separately for each device after the simulation has completed. We list the common parameters used by all devices in Table 5-6. We estimate that links in the on-chip folded torus interconnect are 3mm each. We arrived at these estimates by assuming that the baseline chip would fit in an area of 150mm^2, which represents a die size comparable to published systems of similar design [6]. Formally, the energy of an Orion component is calculated as:

$$E(C_{ORION}, execution) = (P_{Dynamic} + P_{Leakage}) * simulated\_time \qquad (5\text{-}3)$$

5.2.3 Area/Delay Estimates

We use the same two external tools to calculate area and cycle time data. When we report area results in Chapter 6, we report the sum of the areas given by the two tools and make no attempt

to layout the components. Because of this, the absolute area numbers may be misleading and so we present results in terms of relative areas compared to the baseline.

5.2.4 **ASMIX**

All the workloads evaluated run on the ASMIX operating system that was built specifically for ASM-CMP user-mode simulation. ASMIX provides a standard UNIX interface to user software. In this subsection we detail the services that ASMIX provides, and in particular discuss which parts are implemented entirely by ASMIX and which parts have some or all of their functionality emulated.

In the simulator, certain services and/or devices may be emulated without any timing information. The system call mechanisms and associated memory and state management are always modeled in full timing detail, though, meaning that all emulated functions are invoked from guest operating system code. For example, while the simulator emulates a file system service, a file access from user mode will still result in a fully modeled context switch in/out of the kernel, including the code paths needed to copy data from the kernel address space to the user address space.

Table 5-7 lists the system calls provided by the current ASMIX implementation. Additionally, the table shows whether or not that service is ultimately emulated by the ASM-CMP simulator or if it is implemented entirely by ASMIX. All but the final two system calls are POSIX standard calls. The final two are used by ASM-CMP applications for segment management.

Table 5-7. List of ASMIX system calls.

System Call	Emulated?
`read`	Yes
`write`	Yes
`open`	Yes
`close`	Yes
`lseek`	Yes
`fstat`	Yes
`clone`	No
`settls`	No
`exit`	No
`getpid`	No
`waitpid`	No
`create_segment`	No
`modify_segment`	No

5.2.4.2 Pseudo-paging

To evaluate the effect of ASM segmented memory compared to conventional paged memory, we also built in pseudo-paging into ASMIX. The goal of the pseudo-paging implementation is to model the overhead of paged memory assuming that the application under test (a) fits in main memory, (b) is already loaded into memory when the simulation begins, and (c) does not change memory permissions during the run. Under those assumptions, the paging system only needs to manage the TLB and can ignore other, more complicated, aspects of paged memory management like demand paging without losing fidelity. The pseudo-paging implementation maintains a three level page table in software. Instead of a real page table entry at the leaves of the table, the pseudo-pager simply sets a flag to indicate that the page has been "loaded." On a TLB refill (which is software managed in MIPS/ASM-CMP), the refill handler walks the page table and, if it finds the loaded flag set, fills the TLB with a dummy translation. Otherwise it will tell the simulator that it "handled" a page fault, set the loaded flag, and continue on as before.

5.2.5 Software Toolchain

Workload and OS software is compiled and built using a toolchain composed of the **`pcc`** C compiler [191] and a custom assembler/linker. We use version 1.0 of **`pcc`**, which is a venerable tool from the early ages of compilers. It has been retargeted for ASM by starting from an existing MIPS target[10]. In comparison to the MIPS target, the ASM compiler differs in that it emits special load/store instructions for pointer types (see ASM-CMP description in Chapter 4) and modifies the stack management so that the stack grows up. Both of these modifications are used by the initial ASM-CMP prototype but are not required in general for an ASM system.

`pcc` implements basic compiler optimizations, but should not be confused for an industrial strength compiler. **`pcc`** does an admirable job with register allocation, but other forms of optimization are basic at best. For example, it is not uncommon for **`pcc`** to generate a store followed immediately by a load of the same variable even when full optimizations are turned on.

The standard C library used by the simulated applications is a mixture of stock functions extracted from either glibc [192], newlib [193], or fdlibm [194] and custom implementations tailored to ASM. Two portions are worth special mention. First, the pthread library is completely custom, designed both to scale well to 64 cores and to interact correctly with ASM-CMP segments and **`checkout/checkin`**. In the pthread library, mutexes are implemented as ticket locks and barriers use a signal tree developed by Mellor-Crummey and Scott [128].

Second, the ASM standard library implements a custom memory allocator that is similar to a region-based allocator [69]. When an application requests memory, it can specify whether it wants private, acoherent, or coherent memory and the allocator will return a block from a

[10] Readers hoping to use pcc as a MIPS compiler should be wary, as we discovered along the way that it had numerous bugs in that particular target.

segment of the appropriate type. If not specified, the allocator will return acoherent memory. If an application requests more memory than has been pre-allocated to a region, the memory allocator will create a new segment and "link" it to the previous one so that a library checkout or checkin call will still affect all segments in the region.

To keep comparisons fair, the same workloads built for ASM are also used for the coherent runs of the simulator. When doing so, ASM-specific instructions like **`checkout`** and **`checkin`** become no-ops, but otherwise applications for both systems use identical software. The coherent applications still use ASM-CMP segments but the associated types have no effect (i.e., all memory is coherent).

5.3 Workload Selection and Characterization

We select workloads that represent the types of applications that may be run on the baseline system presented above. The workloads are grouped into two general classes. The first class contains existing workloads created for a conventional coherent system and have not been specifically adapted for ASM. The first class is composed of a selection of applications from the SPLASH [158] and SPLASH-2 [180] workload suites. The second class contains workloads tuned specifically for an ASM system. This class contains a variety of workloads that represent different memory usage patterns, including workloads with irregular parallelism. In the following subsections we provide more details about the workloads in each class.

In our evaluation all workloads are run to completion. The workloads are also annotated with markers that indicate when the application's region of interest begins and ends. This typically corresponds to the parallel region of execution. Simulator statistics are reset when the begin marker is encountered so that statistics only reflect execution from a warmed up state. Statistics

are saved upon reaching the end marker so that any cleanup and/or verification code is not included in the results.

5.3.1 Class 1: Unmodified Coherent Workloads

All of the workloads in this class are converted to run on an ASM system using the pthread transformation described in Section 2.5.1. The runtime system creates segments automatically by reusing the segments that exist in C binaries, and assigns segment types based on their semantics (e.g., stacks are private). All applications in this class are parallelized using pthreads. They use a custom version of pthreads that inserts checkout and checkin operations around synchronization calls. The pthread implementation also allocates all low-level synchronization variables (e.g., the integer holding a lock ticket) in a coherent read-write segment type.

With the combination of the automatic segment allocation provided by C and the automatic segment management provided by the pthread library, the workloads in this class work without any substantial modification. Some small changes were made to about half of the workloads. Some workloads shared stack data between threads. Because the conversion mechanism allocates thread stacks in a private segment, such sharing would lead to incorrect execution. In all cases the shared stack data was allocated on the main thread's stack during initialization and then was shared with children that were created later. We fixed this problem by moving the problematic allocations to the heap.

The second correction made in some workloads involved flag variables. These applications used a simple unsynchronized flag to communicate the occurrence of an event such as the creation of a new work. One thread would update the value of the flag to signal peers that spin waiting for the value to change. This pattern will not work as-is because the flags were allocated

Table 5-8. Class 1 Workload Characteristics

Workload	Description	Input	Total Insts (M)	Total Reads (M)	Total Writes (M)
`barnes`	n-body simulation	16K particles	3972.1	799.1	556.3
`fft`	Fast Fourier Transform	1M points	540.6	216.2	79.8
`fmm`	Fast multipole method n-body simulation	16K particles	4602.3	1311.5	569.2
`lu`	Dense matrix triangulation	512 x512 matrix, 16 x 16 blocks	824.5	113.6	55.2
`mp3d`	Wind tunel model	45000	540.7	216.2	79.8
`ocean`	Models ocean currents	258x258 ocean	2283.6	325.8	69.9
`radix`	Integer radix sort	2M integers	252.2	91.4	13.6
`water-nsquared`	Molecular Dynamics	512 molecules	1067.0	193.6	100.9

either on the heap or in the global segment, both of which have the acoherent type. To fix the problem, we converted all unsynchronized flags into synchronized condition variables.

The workloads in this class are a combination of applications from the SPLASH [158] and SPLASH-2 [180] suites. These workloads have been extensively characterized in existing literature [17,158,180], and so we do not belabor the point any further here beyond the information in Table 5-8. The characteristics in Table 5-8, such as total instruction count, differ from those published in the original SPLASH-2 characterization because the architectural and toolchain differences. Previously reported data regarding access patterns, synchronization characteristics, and working sets are still valid, however.

5.3.2 Class 2: ASM-Tuned Workloads

Workloads from the second class were developed to take full advantage of the ASM model. Unlike the coarse segmentation allocation and management used in the first class, we apply a more surgical technique in the second class workloads. In comparison, workloads from the

second class generally use more segments and apply **`checkout/checkin`** only when needed (in contrast to workloads from the first class that often perform useless **`checkin`**s).

Within the second class there are two groups of workloads. The first group of workloads do their own scheduling and data partitioning. The second group are written in a Cilk programming style and rely on a work-stealing task queue for scheduling. The design of the queue used by the second group is explained in detail in Section 2.5.3. In this rest of this subsection we will give an overview of the algorithms used by each workload and provide some high level characteristics.

5.3.2.3 ***Data Parallel Workloads***

heat2d This workload models conductive heat flow over time on a two dimensional surface. Each edge of the surface is kept at a constant temperature and the simulation models how the resulting thermal gradient evolves within the surface. This type of simulation is commonly used to predict how materials will respond to thermal inputs. For example it is used to visualize how a planar slice of magma cools over time after a volcanic event [179]. The physics are modeled with the 2D heat equation from Epperson [57], shown in equation (5-4). For simplicity, in this workload we assume that the surface under investigation is rectangular.

$$\frac{\partial u_t(x,y)}{\partial t} = \alpha\left(\frac{\partial^2 u_t(x,y)}{\partial x^2} + \frac{\partial^2 u_t(x,y)}{\partial y^2}\right) \qquad (5\text{-}4)$$

In equation (5-4) x and y are points in two dimensional space, $u_t(x,y)$ is the heat at point (x,y) at time t, and α is the thermal diffusivity, which is property of the material under investigation. This equation, when discretized using finite differencing for simulation (equation (5-5)), leads to a so-called stencil algorithm because the communication pattern is highly regular (and looks like a stencil when drawn). Each timestep of the simulation uses values from five well

defined points in the previous timestep to approximate instantaneous interaction. To avoid consistency issues that might arise from using data from two timesteps simultaneously, the workload uses two copies of the surface. In a given timestep one copy of the surface (modeling timestep t-1) is read-only and one copy (modeling the current timestep t) is write-only. At the end of timestep the surfaces switch roles.

$$\begin{aligned} u_{t+1}(x,y) = u_t(x,y) \\ &+ \alpha \frac{\Delta t}{\Delta x^2}\left(u_t(x-1,y) + u_t(x+1,y) - 2u_t(x,y)\right) \\ &+ \alpha \frac{\Delta t}{\Delta x^r}\left(u_t(x,y-1) + u_t(x,y+1) - 2u_t(x,y)\right) \end{aligned} \qquad (5\text{-}5)$$

Within a timestep, the update for each point is independent and can be parallelized. We choose to partition the surface evenly between participating threads using the pattern in Figure 2-13 in which each thread gets a complete horizontal slice of the surface. This partitioning has at least two benefits. First, in this stencil algorithm inter-thread communication occurs only occurs on partition boundaries and so to minimize communication the perimeter of the partitions should be minimized. Second, the horizontal partitioning allows us to place each row of the matrix in its own segment, leading to the benefits explained in the case study in Section 2.5.2 (Note that, while we use the same row-partitioning scheme of the case study, this workload uses duplicate matrices rather than checkerboarding to prevent inter-timestep consistency problems). Of course, the horizontal partitioning assumes C-style row-major array ordering. A language that uses column-major ordering like FORTRAN would use vertical partitions.

If the number of segments active at a given time is a concern in an ASM implementation, one could reduce the number of segments per partition (and therefore per thread) to two. The first and last rows of a partition need to be in their own segment because they are communicated at

timestep boundaries. The remaining rows of a partition are always thread-private, and so they can be grouped into a single segment that is used by one thread. With this optimization, the matrix row pointers would point to offsets within one of the two segments.

heat3d This workload also models the thermal characteristics of a material in response to an input, but uses a three dimensional volume rather than a two dimensional surface. The physical model we use is the same as **`heat2d`** but extended with an extra dimension, as shown in equation (5-6). For simplicity, we assume that the volume under investigation is a rectangular prism in this workload.

$$\frac{\partial u_t(x,y,z)}{\partial t} = \alpha\left(\frac{\partial^2 u_t(x,y,z)}{\partial x^2} + \frac{\partial^2 u_t(x,y,z)}{\partial y^2} + \frac{\partial^2 u_t(x,y,z)}{\partial z^2}\right) \quad \textit{(5-6)}$$

Like the two dimension analog, the partial differential equation can be solved using finite differencing, leading to a seven point stencil computation shown in equation (5-8) and graphically in Figure 5-4. Our workload calculates all heat values in a single timestep in parallel. Timesteps are synchronized with a barrier and, like the 2D version, use alternating copies of the volume to avoid consistency issues.

We partition the data statically among threads by making planar cuts through the volume such that each worker gets an equal portion. Under this partitioning, only points in the planes touching a partition need to be communicated between timesteps. To reflect this property, we allocate each plane into a separate segment (the same optimization discussed above to reduce segment bloat can be applied here if needed). At the end of a timestep, workers will check in their boundary planes. At the beginning of a timestep, they check out their neighbor's boundary planes.

$$
\begin{aligned}
u_{t+1}(x,y,z) = u_t(x,y,z) \\
&+ \alpha \frac{\Delta t}{\Delta x^2}\left(u_t(x-1,y,z)+u_t(x+1,y,z)-2u_t(x,y,z)\right) \\
&+ \alpha \frac{\Delta t}{\Delta y^2}\left(u_t(x,y-1,z)+u_t(x,y+1,z)-2u_t(x,y,z)\right) \\
&+ \alpha \frac{\Delta t}{\Delta z^2}\left(u_t(x,y,z-1)+u_t(x,y,z+1)-2u_t(x,y,z)\right)
\end{aligned}
\qquad (5\text{-}7)
$$

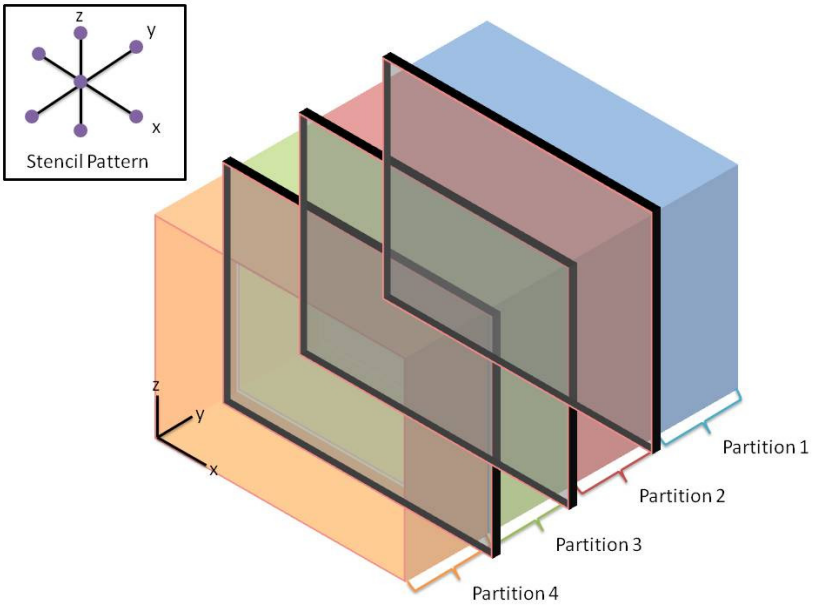

Figure 5-4. Data layout of 3D heat modeling.

Each thread gets an equal partition of the volume. Within each partition, each plane in the x-z dimension is allocated into its own segment. The black slices indicate which planes communicate between timesteps.

5.3.2.4 Task Parallel Workloads

All the task parallel workloads use the Cilk-like task scheduler discussed in the case study of Section 2.5.3. The first four are taken directly from the Cilk source release [195]. Because we did not implement the compiler features of Cilk that allow scheduling to be communicated with language-level keywords, we modified the source so that they use the API in Figure 2-15. The modifications are straightforward, and involve, for example, replacing the Cilk `sync` keyword

with a call the function **`taskq_sync()`**. All of these workloads use the Cilk fork-join programming style. The final workload, **`uts`**, was taken from the release of HotSLAW [129], a task queue implementation for the UPC language [33].

matmul The **`matmul`** program is a dense matrix multiply application. It is a Cilk implementation of the REC-MULT algorithm notable for being provably *cache-oblivious*. [66]. The algorithm proceeds in a recursive divide and conquer manner according the cases in equation (5-8).

$$\begin{aligned}
\begin{pmatrix} A_1 \\ A_2 \end{pmatrix} B &= \begin{pmatrix} A_1 B \\ A_2 B \end{pmatrix} && \text{if} \quad m \geq \max\{n, p\} \\
\begin{pmatrix} A_1 & A_2 \end{pmatrix} \begin{pmatrix} B_1 \\ B_2 \end{pmatrix} &= A_1 B_1 + A_2 B_2 && \text{if} \quad n \geq \max\{m, p\} \\
A \begin{pmatrix} B_1 & B_2 \end{pmatrix} &= \begin{pmatrix} AB_1 & AB_2 \end{pmatrix} && \text{if} \quad p \geq \max\{m, n\}
\end{aligned} \tag{5-8}$$

Because all cases subdivide the problem in half, all tasks in the **`matmul`** workload that are not leaves will have exactly two children. In Figure 5-5 we show the task dependency graph of a **`matmul`** execution that multiplies two 512x512 matricies using eight processors. Each color represents a worker thread, and each task is assigned the color of the worker that executed it. Because the task dependencies form a deep graph there is very little work stealing in the **`matmul`** application.

In the **`matmul`** application, the two input matricies are read-only after initialization, and as such they are each allocated into a coherent read-only segment. The output matrix is write-only and is allocated into its own acoherent segment. Because the tasks in the **`matmul`** algorithm are completely independent, segment management is embarrassingly simple (aside from the

management occurring in the task runtime). There is only one static checkin operation that is performed on the output matrix when a leaf task completes.

lu This workload decomposes a dense matrix *A* into upper and lower triangular components such that A = LU, shown for an example 2x2 matrix in equation (5-9). An LU decomposition is the starting point of many algorithms, including linear system solvers. The algorithm used in `lu` is recursive. It starts by factoring A_{00} into $L_{00}*U_{00}$ and then simultaneously uses back substitution to solve for U_{01} and forward substitution for L_{10}. More details on the algorithm can be found in Randall's dissertation [150].

$$\begin{bmatrix} A_{00} & A_{01} \\ A_{10} & A_{11} \end{bmatrix} = \begin{bmatrix} L_{00} & 0 \\ L_{10} & L_{11} \end{bmatrix} \bullet \begin{bmatrix} U_{00} & U_{01} \\ 0 & U_{11} \end{bmatrix} \qquad (5\text{-}9)$$

Figure 5-7 shows the task dependency graph produced by a run of `lu` on a 128x128 matrix and eight threads. LU is composed of relatively short tasks and wide dependency tree. Most tasks have a fan out of four, which results from the recursive steps in the algorithm that partition the root matrix into four equal submatricies. In part due to the shallow structure of the task dependency graph, the number of task steals is relatively high in the lu application.

multisort The `multisort` application performs an integer sort on a one dimensional array. It is similar in structure to a common mergesort but performs each merge using a variation of binary search that avoid memory conflicts. It was first developed by Akl and Santoro [9] and was later adapted to Cilk by Frigo and Stark [59]. The application partitions work into tasks until the partition size is less than 2K, at which point it performs a serial sort. The task dependency graph for a multisort of 10,000 integers and eight threads is shown in Figure 5-6. The graph is

relatively wide, signifying an abundance of parallelism. The regularity of colors in the graph also indicates that there is little work stealing throughout the application.

uts The `uts` application is a synthetic unbalanced search tree algorithm created by Olivier, et al. [142]. It is designed to evaluate systems intending to run task-based graph algorithms. It performs a parallel traversal of an irregular and unpredictable search space. The tree is generated on the fly by probabilistically creating children nodes according to a distribution specified as an input. In our evaluation, we use two different inputs. First, in the workload labeled `uts_circ`, the tree is created with a cyclic geometric distribution. The `uts_circ` tree contains 4117769 nodes, has a depth of 81, and contains 2342762 leaves (56.89%). It corresponds to input T2 in the UTS distribution. The `uts_fixed` workload follows a fixed geometric distribution corresponding to the T1 input. The `uts_fixed` tree contains 4130071 nodes, has a depth of 10, and has 3305118 leaves (80.03%).

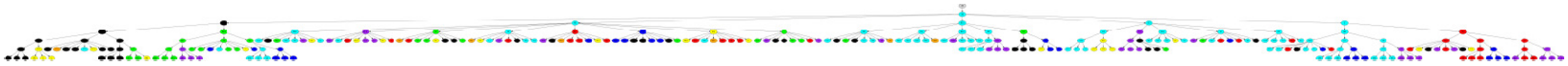

Figure 5-7. `lu` **task dependency graph.**

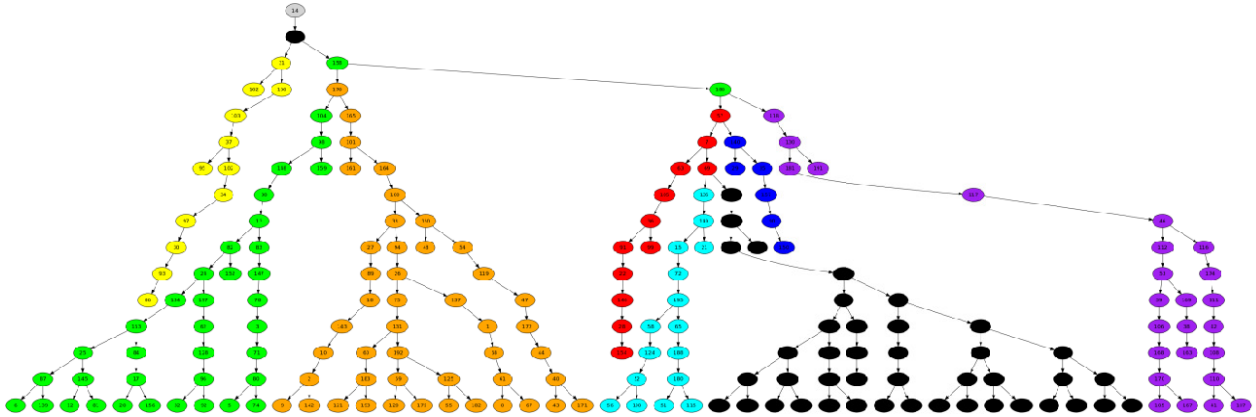

Figure 5-5. `matmul` **task dependency graph**

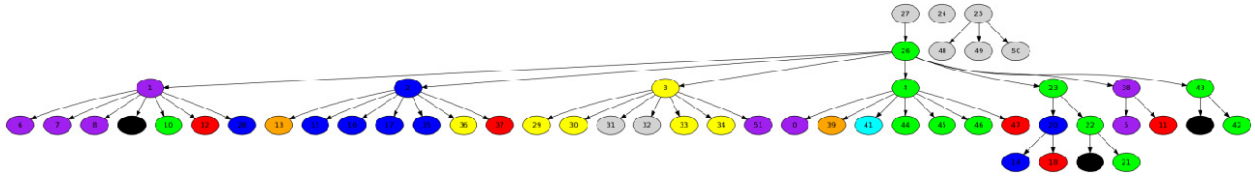

Figure 5-6. `multisort` **task dependency graph**

6

Evaluation

In this chapter we present an evaluation of ASM-CMP compared to the conventional cache coherent baselines described in Chapter 5. Overall, we show that ASM-CMP is a more simple design than conventional coherent hardware, and that the simplicity does not sacrifice either performance or power. We find ASM-CMP performs the same as the cache coherent baselines, and in some cases (e.g., in the presence of false sharing) much better. Because ASM-CMP does not use a coherence directory, it requires less on-chip memory energy compared to conventional CMPs, though overall (system wide) the energy is comparable.

We also investigate the versatility of ASM-CMP and show that it is able to adapt well to a wide range of sharing patterns. We highlight the inefficiencies in the ASM-CMP design by showing how much performance could be gained with hardware that uses oracle information about acoherence operations. In Section 6.6 we also show the benefits of ASM-CMP segmentation compared to conventional paging in terms of both performance and energy.

In Section 6.5 we also provide a detailed characterization of the ASM workloads. We show that class-1 and class-2 workloads differ in the way they use checkout and checkin instructions, and provide intuition on why that is. Our characterization shows that both classes of workloads perform significant work during checkout and checkin (e.g., flush up to one half of the L1 data cache on a single checkin). This point underlies the importance of being able to hide the latencies of the checkout and checkin operations by making them non-blocking instructions.

Finally, since most of the results presented in this chapter are normalized for easy comparisons, we provide the raw data in table form in the accompanying Appendix C.

Table 6-1. Complexity comparison of ASM-CMP to cache coherence.

	ASM-CMP	MESI	MOESI
Stable L1 States	3	4	7
Transient L1 States	5	6	8
Stable L2 States	2	3	13
Transient L2 States	3	14	46
Total States	13	27	54
State Overhead	2.2%	11.9%	20.2%

6.1 Simple Hardware

We compare the complexity of ASM-CMP both qualitatively and quantitatively. Qualitatively, we observe that ASM-CMP, while still using a finite state machine (acoherence engine) to control caches, uses a more localized design and associates state with private, rather than shared, caches. In particular, in ASM-CMP cache controllers only accept local, processor-side requests, and as a result do not have to deal with distributed race conditions. Additionally, because ASM-CMP allows multiple writers, cache controllers do not need to serialize access to a block, which can be another source of complexity in coherence controllers, especially in controllers for shared last level caches.

Quantitatively, we compare ASM-CMP to both cache coherence protocols by comparing the number of states required by their controllers, shown in Table 6-1 (repeating information found in Table 5-3). ASM-CMP requires less than half the number of states needed by the inclusive MESI protocol and less than a quarter of the states required by the more complicated MOESI protocol. Additionally, because ASM-CMP only adds state on the (smaller) private data caches, rather than shared unified caches, it has 5x and 10x less state overhead, respectively.

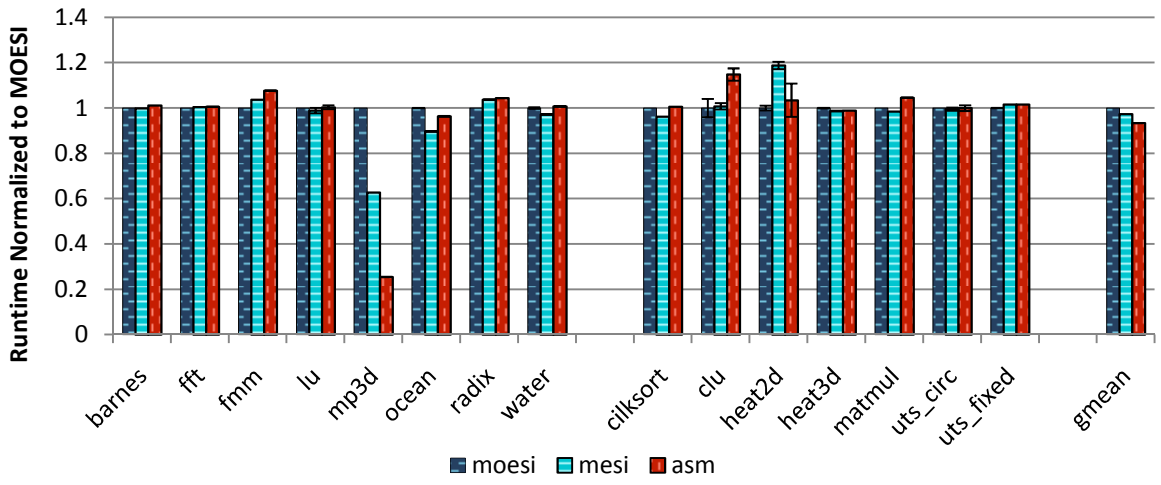

Figure 6-1. Runtime of ASM versus cache coherent baselines.

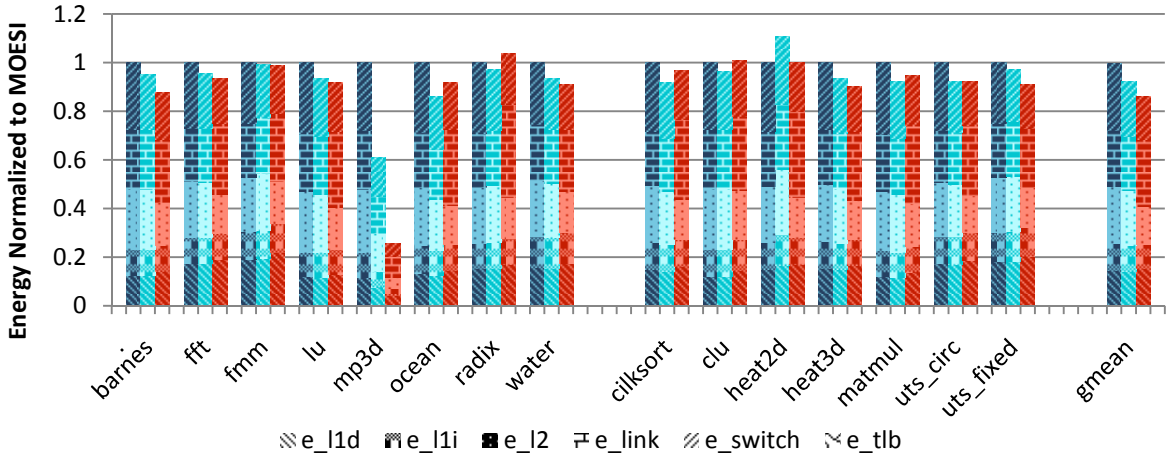

Figure 6-2. Energy of ASM versus cache coherent baselines.

6.2 Comparable Performance and Energy

We show the performance of all workloads on ASM-CMP and both coherent baselines in Figure 6-1. In the graph, all bars are normalized to the performance of the MOESI baseline for that particular workload. Overall, we see that ASM performs comparably to both coherent configurations. In `mp3d`, ASM outperforms both the MOESI (non-inclusive) and MESI (inclusive) protocols by approximately 4x and 3x, respectively. This is due almost entirely to the fact that ASM eliminates false sharing, which is prevalent in `mp3d`. There is also a secondary

effect from a high degree of true sharing in a migratory pattern, which **mp3d** also does *en masse* and which ASM-CMP can perform with greater efficiency than invalidation-based coherence. The MESI protocol performs better than MOESI because the timing of requests in that protocol happen to result in many fewer forwarded requests. In MESI, for heavily contended blocks there are more accesses on average between invalidations. This ultimately leads to better performance because a processor is able to make better use of the block before it is given up to a remote thread, similar to the effect of delayed invalidations [113].

In **fmm** and **clu**, ASM performs worse than the coherent baselines. These programs frequently check-out data that they are about to use again and that has not been modified by a remote thread. Thus, in both workloads there are unnecessary L1 cache misses that negatively affect performance. In Figure 6-3 we show that without the unnecessary misses all of the lost performance is regained.

Figure 6-2 shows the on-chip memory energy consumed by all workloads. We find that ASM-CMP always consumes less on-chip memory energy (average 18%), even in cases where the performance was slightly worse. Most of the reduction is due to the absence of a directory in the L2 cache. In a system like ASM-CMP, the energy of the L2 is dominated by static leakage power, and so reducing area leads to substantial energy benefits.

In Table 6-2 we compare the hardware overheads required by all three systems. ASM-CMP uses less state in the baseline configuration. In ASM-CMP the overheads are associated purely with the private cache storage, and so it will scale well in larger systems where the amount of shared memory grows faster relative to the amount of private memory. Of course, the inverse is also true.

Table 6-2. Comparison of state overheads to manage data sharing. All values are in kilobytes (KB).

Component	MOESI	MESI	ASM
Metastate Lookaside Buffer			32
Dirty Byte Bitmask			64
Sidecar Registers			8
L1 Coherence States	32	32	
L2 Directory States	1024	1664	
Total	**1056**	**1696**	**104**

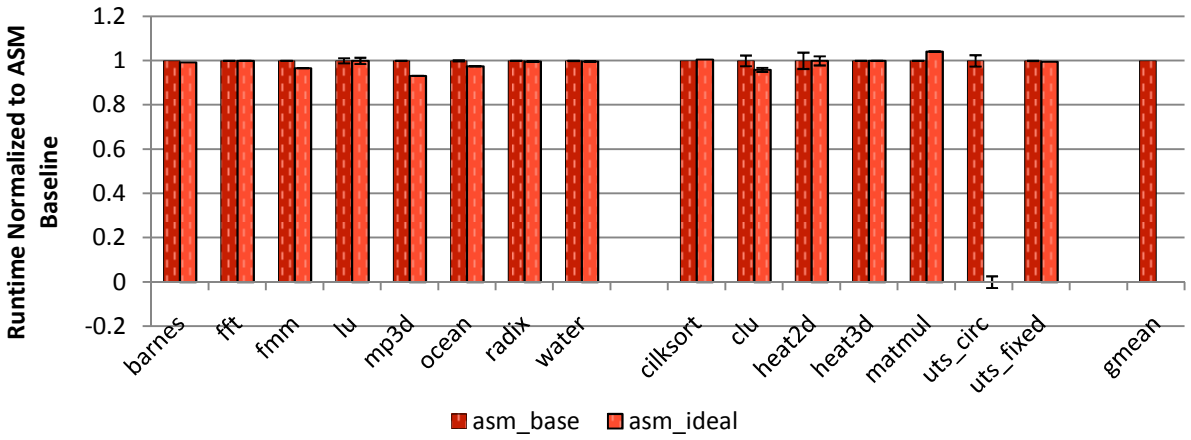

Figure 6-3. Performance impact of a perfect checkout operation.

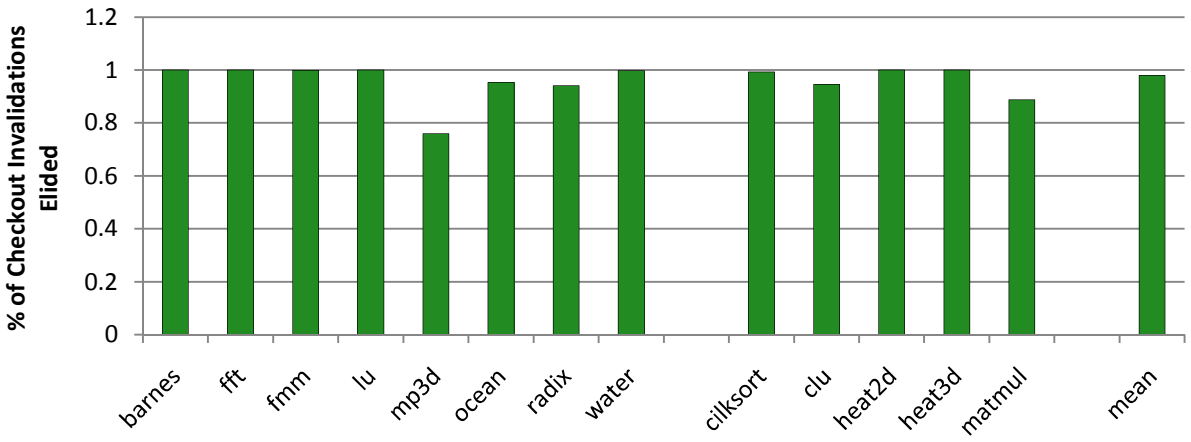

Figure 6-4. The percentage of block invalidations that are elided by a perfect checkout.

6.3 Performance Analysis

In this subsection we try to provide insight on the performance of ASM-CMP and show how future systems might improve on the design.

In Figure 6-3 we show the performance of an ASM-CMP system that uses oracle information to avoid useless invalidations on checkout operations. In these runs, a block is invalidated by a checkout only if it has also been modified remotely since the previous checkout occurred on that thread. When compared to the baseline performance in Figure 6-1, we find that overheads from conservative checkout operations account for nearly all of the performance difference when ASM execution is slower than cache coherence. Thus, the checkout operation would be a good optimization target for future iterations of the ASM-CMP design.

We shed more light on the behavior of checkouts in Figure 6-4, which shows how many invalidations are elided when using the perfect checkout mechanism described above. On average, over 97% of invalidations are skipped when performing a perfect checkout. Because performance is normally unaffected by these elisions, we conclude that in most cases the unnecessary invalidations occur on blocks that will not be referenced again in the near future. In other words, the invalidations that happen during checkout usually occur on blocks that will be replaced before they are accessed again. Intuitively, this means that when applications check a segment out, they have moved on to another phase of computation and will not access the data they recently touched.

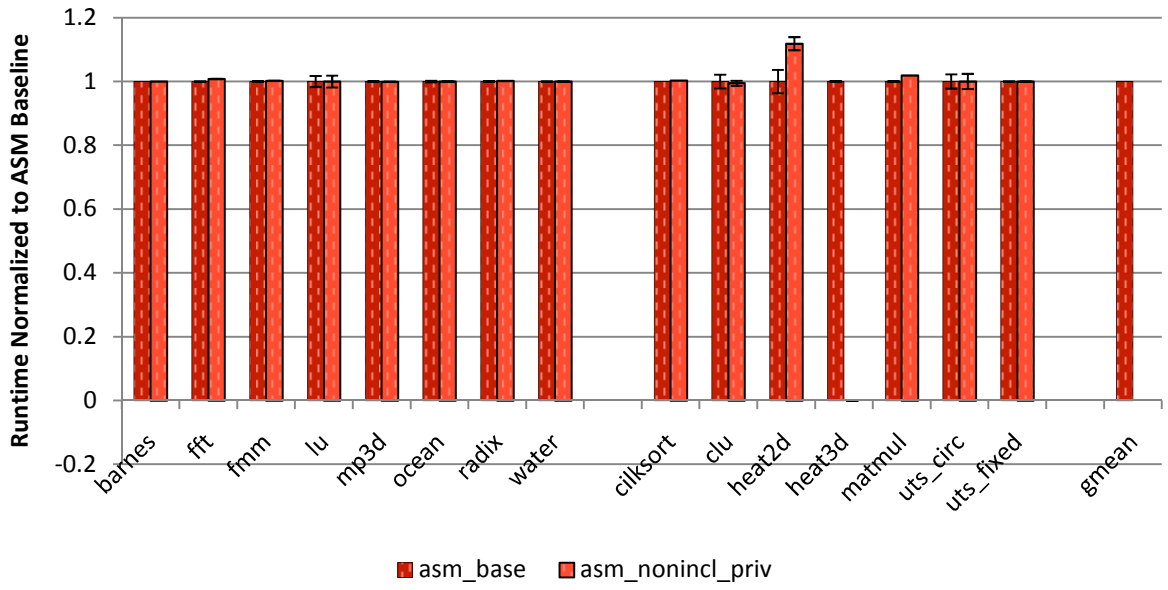

Figure 6-5. Performance impact of private data differentiation.

In Figure 6-5 we show the effect of ASM-CMP's private data policy. In the bar on the left, we show the baseline ASM-CMP runs. In the bar on the right, we show the performance of an ASM-CMP system that uses a non-inclusive, rather than exclusive, policy to manage private data. Most workloads are statistically unaffected, but some, including heat2d, see a performance improvement from the policy. This is unsurprising, as **`heat2d`** is the *only* workload that makes heavy use of private segments other than the stack. We suspect that if we were to finely tune other workloads to identify private allocations, especially in class-1, that we would find noticeable performance benefits.

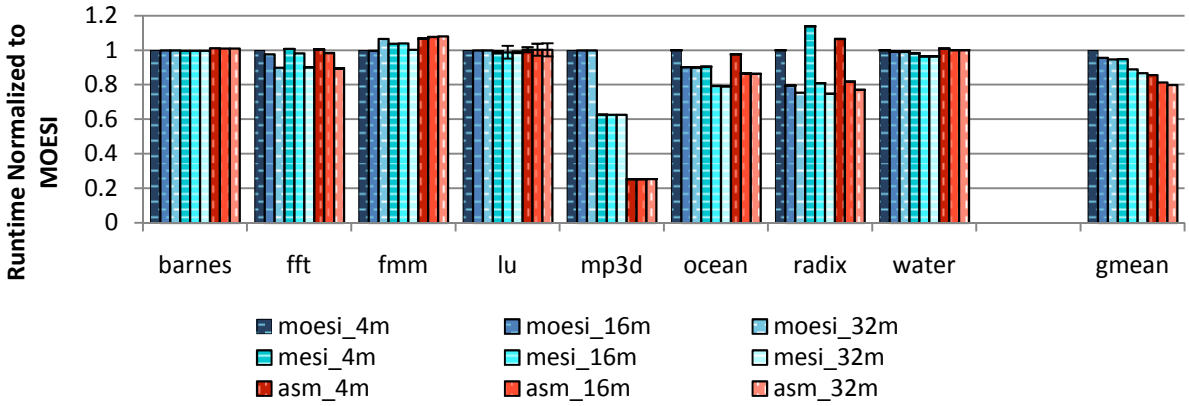

Figure 6-6. L2 cache size sensitivity analysis for class-1 workloads.

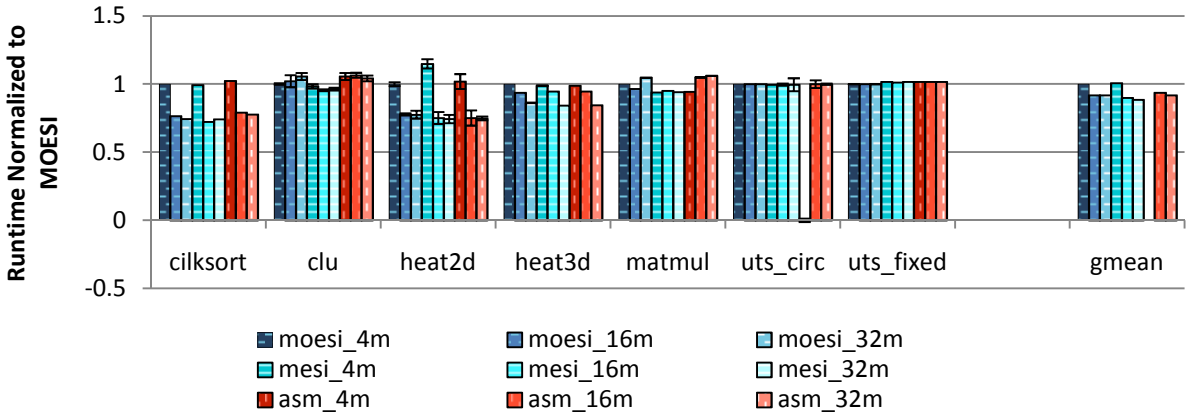

Figure 6-7. L2 cache size sensitivity analysis for class-2 workloads.

6.4 Cache Sensitivity Analysis

In Figure 6-6 and Figure 6-7 we show the performance of all three systems as the aggregate size of the L2 cache is scaled from 2MB to 32MB. When scaling, we did not alter the access time of an L2 bank. The results show that ASM-CMP is affected by the aggregate L2 size similarly to the non-inclusive MOESI protocol. Compared to the inclusive MESI protocol, ASM-CMP in some cases scales better. For example, in **`radix`** and **`heat2d`**, ASM-CMP performs better than MESI with a small L2 cache and comparably with a large L2 cache. We find this is because ASM-CMP is able to better utilize the cache capacity because it does not waste resources maintaining inclusion [186].

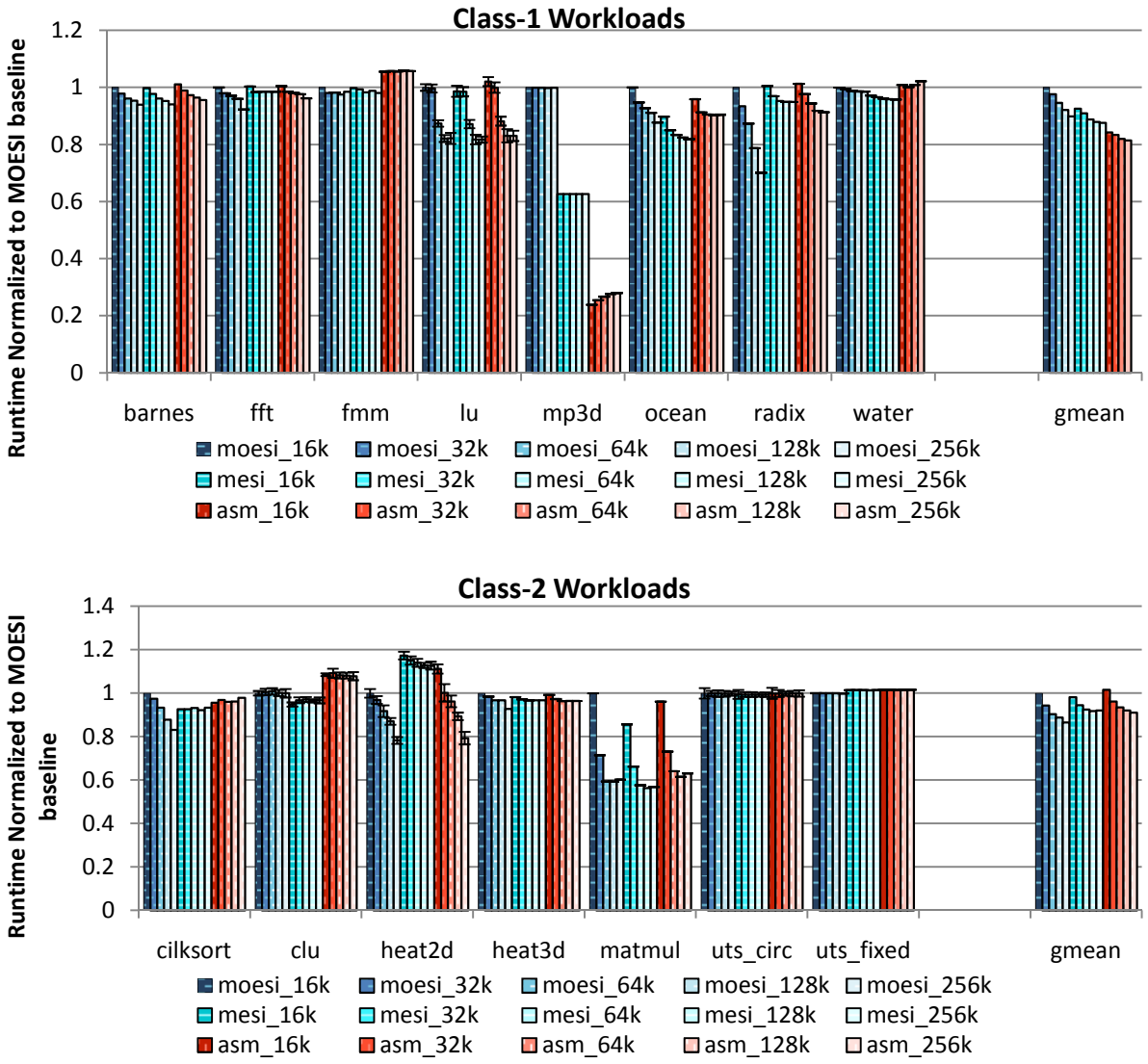

Figure 6-8. L1 cache size sensitivity analysis.

In Figure 6-8 we show how performance is affected by the L1 data cache size. First, we notice that in most cases ASM-CMP scales similarly to the coherent baselines. In some cases, e.g., water, ASM-CMP actually performs worse as the size of the L1 increases. This occurs because in ASM-CMP larger private caches make the overhead of checkout and checkin larger, negating some of the benefit of an increased capacity cache. We believe that some of these overheads could be negated with the techniques described in Chapter 7.

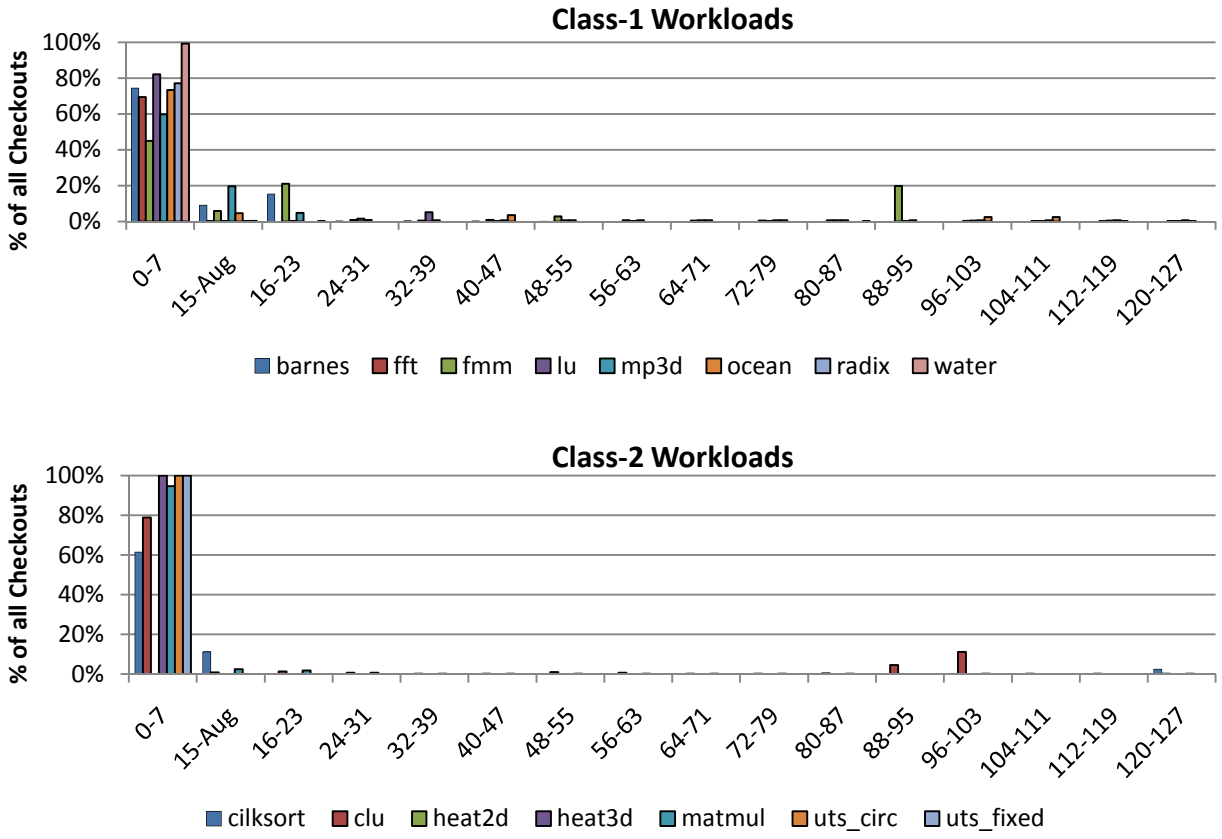

Figure 6-9. Histogram showing how many blocks are invalidated by checkouts. In these graphs we exclude checkouts that affect zero blocks.

6.5 Workload Characterization

In this subsection we show the characteristics of checkout and checkin operations.

6.5.1 Checkout Characteristics

In Figure 6-9 we show a histogram of the number of segment blocks invalidated by a checkout operation. The graphs exclude checkouts that do not invalidate any blocks (more on this below). The X axis shows the number of affected blocks and the Y axis shows the percentage of all checkouts that fall into that bin. For reference, we provide the absolute number of checkouts performed by each workload in Table 6-3. The graphs are truncated at 128 blocks for clarity.

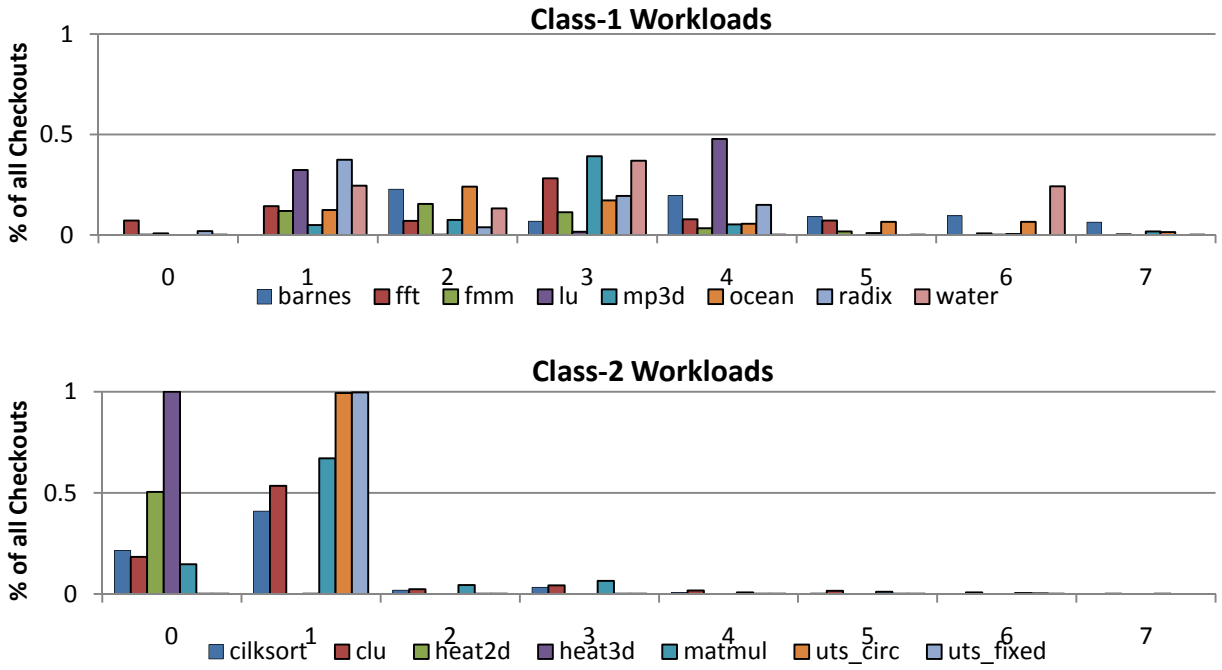

Figure 6-10. Checkout invalidations when the number of affected blocks is less than 8.

The majority of checkouts are small, affecting 7 blocks or less. Larger operations do occur, typically correlated with synchronization events (e.g., upon leaving barrier). Some operations, such as those in `fmm` and `clu`, invalidate up to 25% of the L1 data cache. It is these larger events that cause the performance loss corrected by the oracle invalidation scheme discussed above.

In Figure 6-10 we show the same information but limit the bins to values 0-7 and include checkouts that do not affect any blocks. We find that most workloads affect between 1-4 blocks when checkouts are small. We also see from these graphs a difference between class-1 and class-2 workloads. In class-2 workloads, there are a significant number of checkouts that do not affect any blocks, i.e., when a thread checked out there were no valid blocks in the L1 data cache. These checkouts occur because of the nature of task-driven execution; at many steps a task cannot be sure if it has already touched data because it may not have a history of the tasks already completed. Their presence suggests that future implementations may want to optimize these empty checkouts, perhaps by skipping timestamps when there is nothing to do.

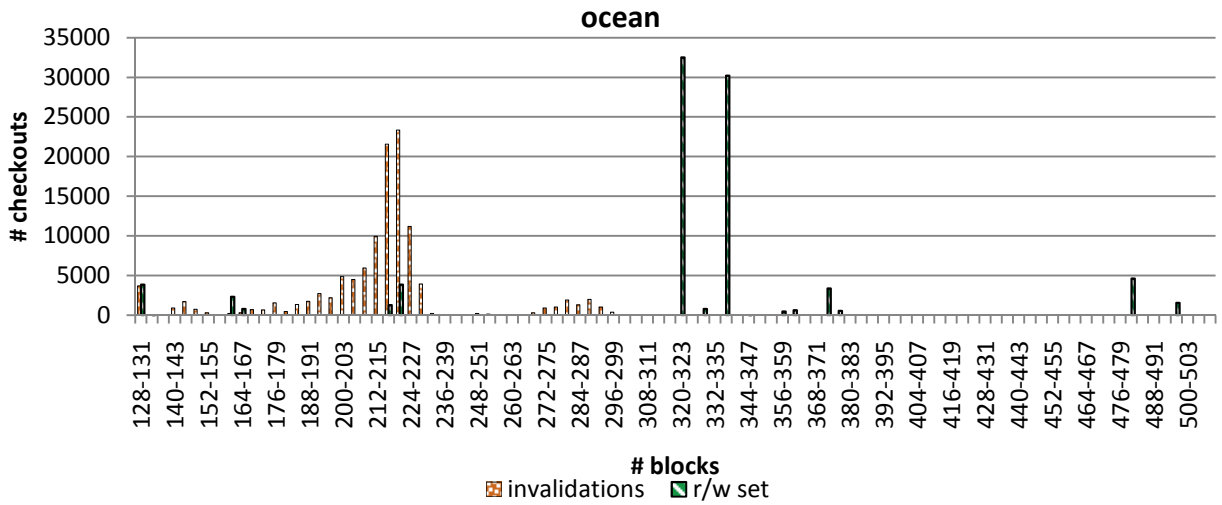

Figure 6-11. Absolute number of blocks invalidated and touched between checkouts in ocean.

In Figure 6-11 we show both the absolute number of blocks invalidated (invalidations) by checkout and the absolute number of blocks touched (r/w set) between checkouts for the **ocean** workload. The difference between the two represents the blocks that are evicted from the L1 before a checkout due to capacity constraints. We show only the larger bins, between 128-512 blocks, to show that when the checkout region is large many blocks are evicted before checkout. In particular, the graph shows that when the number of blocks touched is around 330 (green cluster), only about 220 of those (orange cluster) are invalidated, and thus approximately 100 blocks are evicted prior to checkout.

ocean is an average workload; some perform even more silent evictions and others perform almost none. In Table 6-3 we show the absolute total number of acoherence related events for all workloads. The number of silent evictions for a workload can be seen as the difference between the number of blocks invalidated and the number of blocks in the read/write set.

Table 6-3. Checkout/Checkin characteristics.

 # invals is the number of blocks invalidated by a checkout.
 # blocks in r/w set is the number of blocks touched between consecutive checkout operations.
 # flushes is the number of blocks flushed to the L2 by checkin operations.
 # blocks in write set is the number of blocks written between consecutive checkin operations.

Workload	# checkouts	# invals	# blocks in r/w set	# checkins	# flushes	# blocks in write set
barnes	186251	1345983	1567602	186095	310997	382142
fft	892	113243	2590017	770	45996	786435
fmm	355602	11224561	11464970	355492	1198442	1286960
lu	8574	233008	1101350	8450	160670	364100
mp3d	149508	3971088	4876603	149382	3439199	4334167
ocean	109172	4124549	8425990	109058	1863249	3998279
radix	3263	247235	1025798	2924	94937	387041
water	139267	550726	691855	439234	91699	220237
cilksort	55317	2490545	3775521	29374	1477900	18174672
clu	78576	1173997	2253381	56701	555060	14212094
heat2d	12600	798336	3193344	12600	806400	3225600
heat3d	8448	1	16850945	4224	540441	8688768
matmul	175317	342123	1470902	106482	427433	24811746
uts_circ	4145521	4263691	5365945	98515	1731064	1194812467
uts_fixed	4142819	4204508	5265893	41416	1043064	478970618

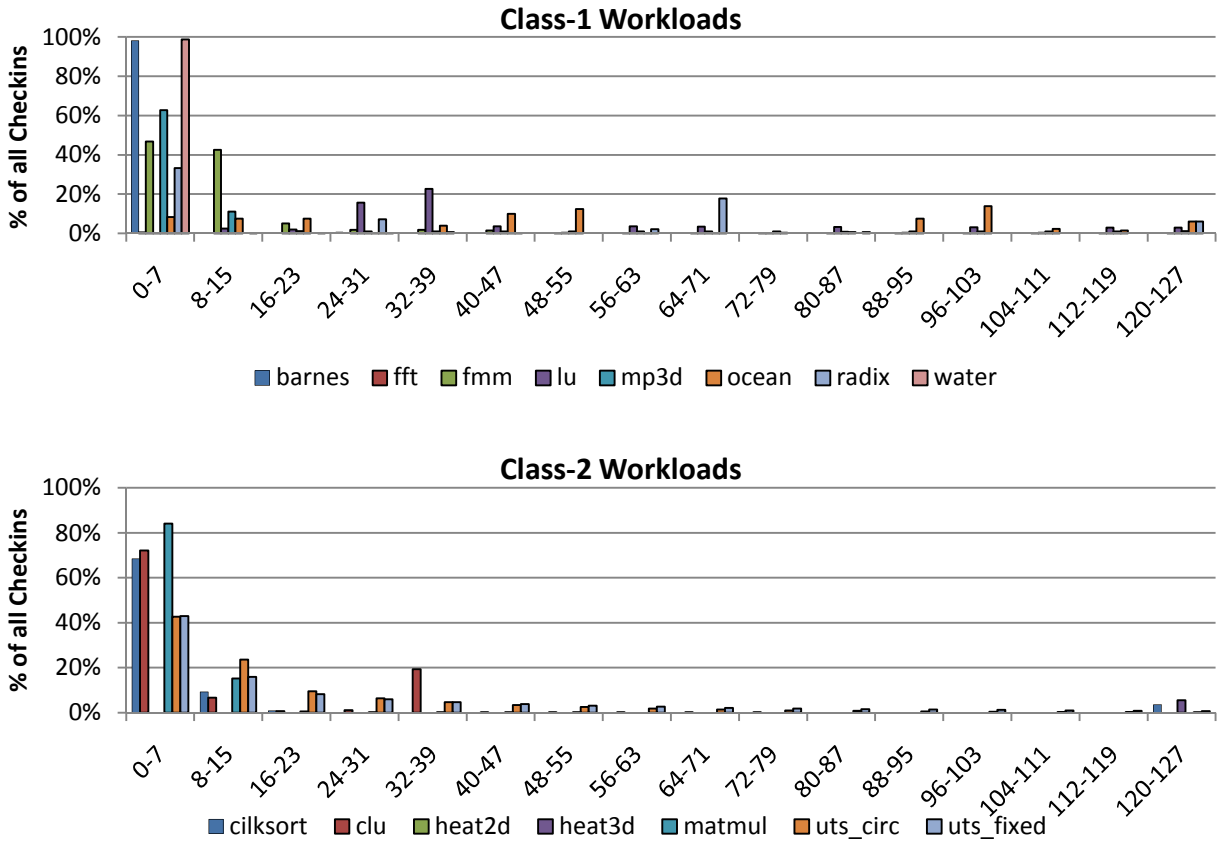

Figure 6-12. Histogram showing the number of blocks flushed by checkin operations. In this figure we exclude checkins that flush zero blocks.

6.5.2 Checkin Characteristics

In Figure 6-12 we show a histogram of the number of blocks flushed by checkin operations. The graphs exclude checkins that do not flush any blocks. The X axis contains bins representing the number of blocks flushed per operations, and Y axis shows the number of checkins that fall into those bins. Graphs are truncated at 128 blocks for clarity.

The graphs show that the majority of checkin operations affect a small number of blocks. There are still significant numbers of checkins that flush a large portion, e.g., 25%, of the L1 data cache. We find that in most cases the latency of the checkin operation is overlapped with synchronization delay (e.g., a checkin followed immediately by a barrier), and thus they do not impact performance. These results highlight the benefit of non-blocking checkins.

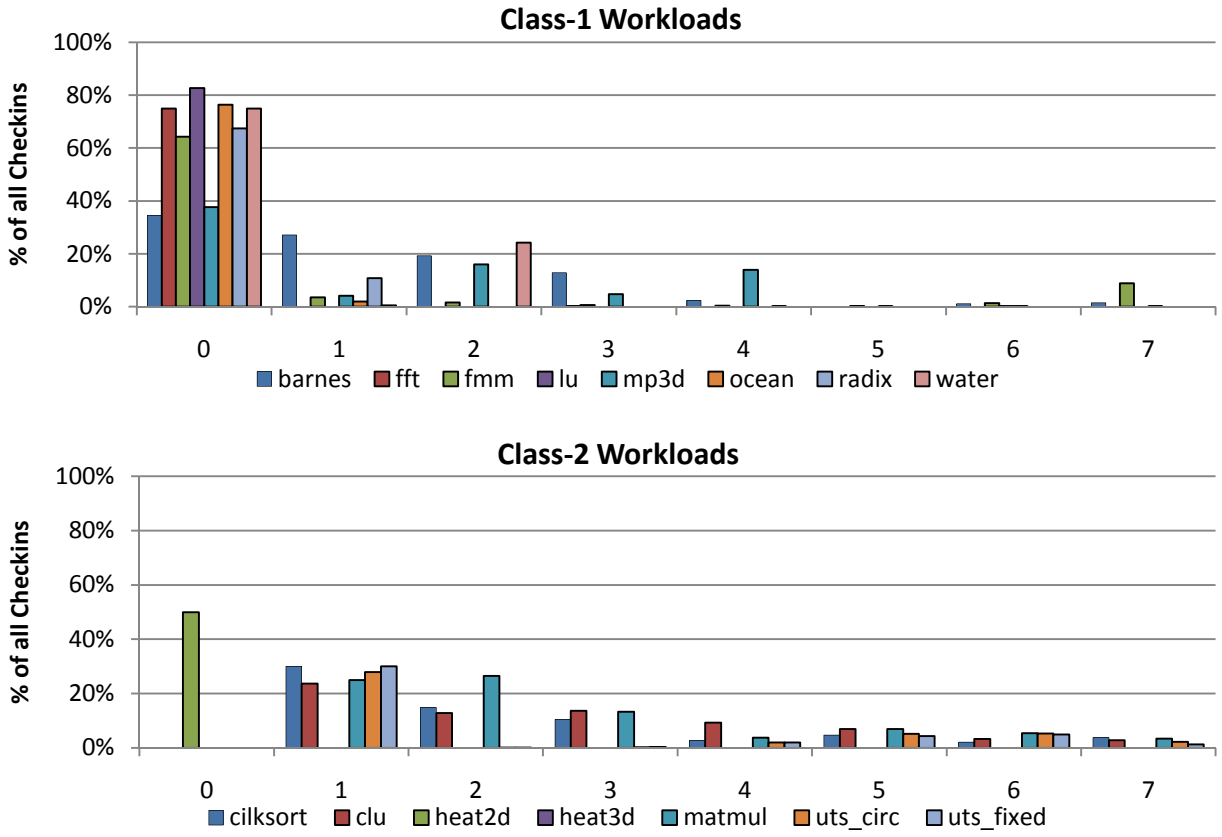

Figure 6-13. Checkin flushes when there are less than 8 affected blocks.

In Figure 6-13 we show a histogram of checkin flush set size when there are less than 8 affected blocks. We find that class-1 workloads perform a large number of checkins that affect zero blocks. These arise from the conservative nature of the automatic transformation described in Section 2.5.1. It is common, for example, in these workloads to check in the global segment right before a synchronization event when all the updates were actually just performed on the heap (in a separate segment).

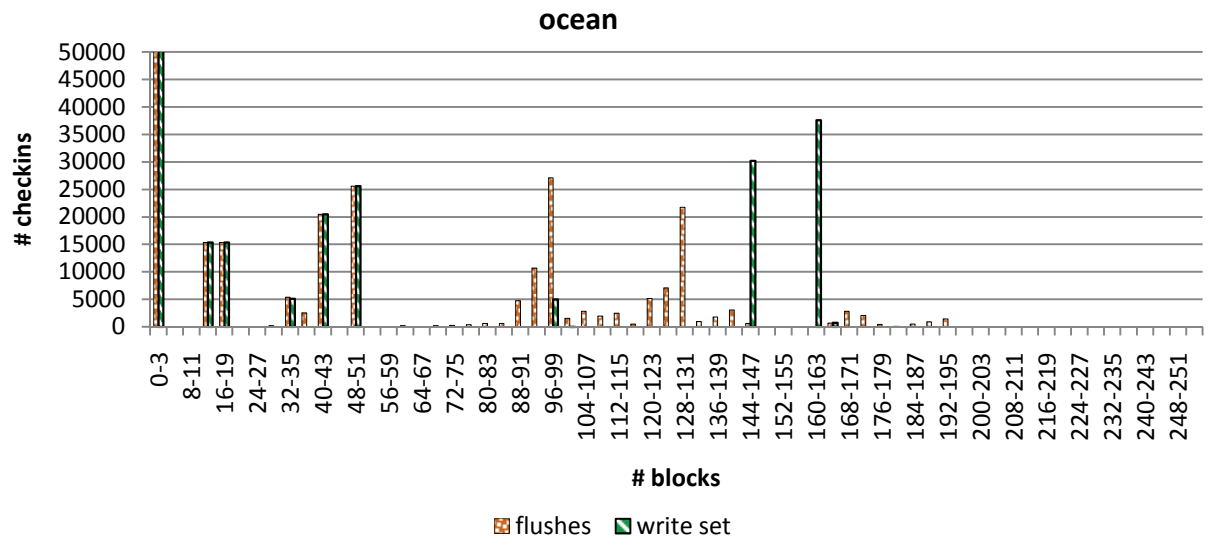

Figure 6-14. Number of blocks flushed and written, respectively, between consecutive checkins.

To analyze checkin characteristics further, we again choose **`ocean`** as a representative workload. In Figure 6-14 we show both the number of blocks flushed by checkins and the number of blocks written between consecutive checkins. The difference between the two sets is the number of blocks evicted early from the L1 cache prior to being checked in. We find that when the number of written blocks is large (e.g., > 80), a significant portion of the write set is evicted early. This would seem to indicate that 80 blocks is around the sweet spot for applications using best-effort acoherence; regions that write more than 80 blocks would probably fail frequently, resulting in poor performance.

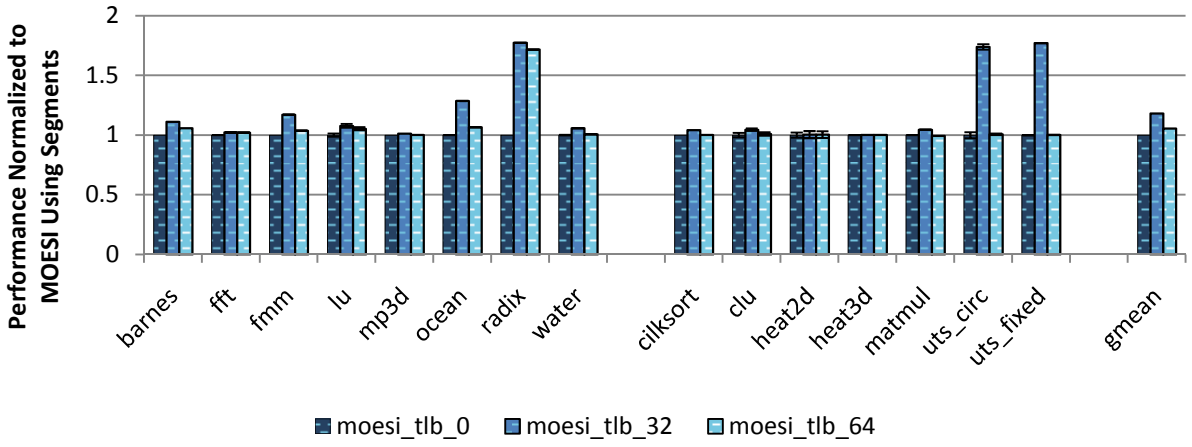

Figure 6-15. Performance of MOESI baseline with a varying sized TLB.

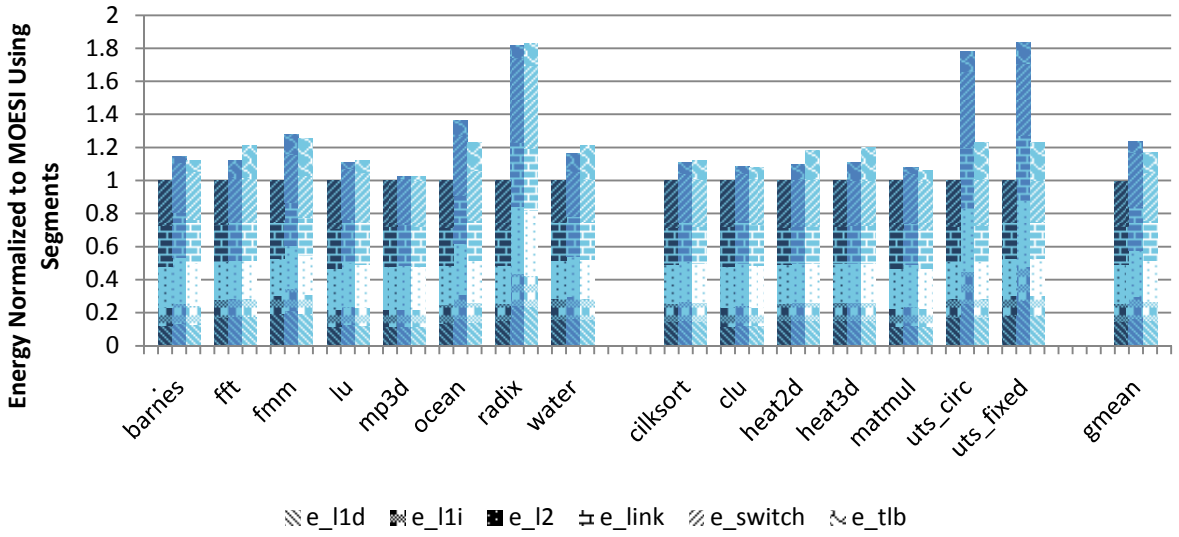

Figure 6-16. On-chip memory energy of MOESI baseline with a varying sized TLB.

6.6 ASM-CMP Segmentation Benefits

We conclude our evaluation of ASM-CMP by analyzing the effect of ASM-CMP segmentation compared to conventional systems that use paging. In Figure 6-15 we show the performance of the baseline MOESI system using ASM-CMP segments and the pseudo-paging mechanism described in Chapter 5 that for our workloads closely approximates true paged virtual memory. We compare against two TLB sizes of 32 and 64 entries, respectively.

The results indicate that for our workloads, ASM-CMP segmentation can improve performance on average 18% and 5% for a 32-entry and 64-entry TLB, respectively. Because we do not model page faults (and strongly suspect these workloads would not incur them anyways), the slowdown from paging can be attributed entirely to the latency required to walk the page table on TLB misses. In most of the evaluated workloads, a 64-entry TLB is large enough to eliminate most of the performance loss, but, as we show in Figure 6-16, that comes at a cost of increased energy relative to ASM-CMP segmentation. For example, in **`uts_circ`** the performance gained by a 64-entry TLB comes at the cost of 20% extra energy.

7

Conclusions and Future Work

In this chapter we summarize the primary findings of this dissertation and present future directions motivated by those findings. In particular, we present future directions in four general categories. In Section 7.2.1 we discuss how the ASM model might be extended even further to benefit both hardware and software. Then in Section 7.2.2 we present ideas for how the ASM model might be scaled back closer to a conventional system while still maintaining most of the benefits. In Section 7.2.3 we discuss future optimizations specific to the ASM-CMP prototype. Finally, in Section 7.2.4 we present ideas for how software might exploit the ASM model even further than the ideas already presented in this dissertation.

7.1 Conclusions

In this dissertation we have presented ASM as a new shared memory model for commodity multiprocessors. We have shown the benefits of ASM over coherent shared memory, including the ability to communicate data semantics between hardware and software, and the ability for software to represent the semantics of the synchronized data in particular. We performed four case studies to show that the ASM model is easy for programmers to use, and can even permit algorithms not possible in coherent shared memory. We have formalized the model for a deeper understanding of its implications and possibilities. We showed that the model permits implementations to perform relaxed acoherent operations and that the model is equivalent to sequential consistency for well formed programs. Our initial prototype, ASM-CMP, is able to exploit the properties of the ASM model to provide performance on par with existing coherent systems while requiring less state and less energy. We have also made an argument that ASM-CMP memory system is less complex than cache coherent systems, mainly because the ASM-CMP cache controllers only need to consider local, processor-side, requests and never deal with remote requests or the race conditions that arise from them.

We conclude that the ASM model is a plausible design point for future multiprocessors seeking energy efficiency and reduced complexity. We believe ASM holds promise as a model for future heterogeneous systems that use accelerators that either do not want or need full cache coherence and can lower the barrier to entry for custom components.

7.2 Future Directions

7.2.1 ASM Model Extensions

Using Checkout/Checkin for Synchronization. The checkout and checkin operations are almost exclusively associated with synchronization events, yet they are not synchronization operations themselves. This can lead to inefficiencies in an implementation. For example, in ASM-CMP, when a thread checks out, the acoherence engine will use atomic memory operations to transparently acquire a segment timestamp. If that checkout occurs after a synchronization event that itself uses atomic instructions, then the timestamp acquisition is in a sense redundant.

```
barrier state:
  Timestamp cur_ts;           // can be thread local
  Segment protected_segment;  // can be thread local


barrier_init:
   cur_ts = checkout (protected_segment)


barrier_wait:
  checkin(protected_segment)
  cur_ts = cur_ts + num_threads
  do {
     co_ts = checkout(protected_segment)
  } while (co_ts != cur_ts)
```

Figure 7-1. Barrier implementation that uses acoherence timestamps.

We might be able to eliminate these inefficiencies by extending the ASM model to expose the ordering properties of checkout and checkin to software by having the instructions return their timestamp. For example, in Figure 7-1 we show how software could build a barrier primitive that uses the timestamps of checkout operations. Notice that this barrier does not need to use any conventional atomic instructions; rather, the atomic operations are implicit in the checkouts. This

example points to another possible model improvement, namely blocking acoherent operations. If a checkout operation could block until, for example the segment timestamp reached a particular value, the barrier in Figure 7-1 could avoid spinning (one can think of this as giving checkout monitor/mwait functionality).

Timestamps, as implemented by ASM-CMP, only contain information about a particular segment. If timestamps were used for more general synchronization, it may be helpful to make them true Lamport Scalar Clocks [107] so that software could use them to reason about all segments in a process. In such a design, timestamps would be updated to reflect the relationship between disjoint segments on the same thread.

Segment Remapping. In some programs it could be helpful to remap segments dynamically between phases. For example, in a divide and conquer sorting algorithm like mergesort, it could be helpful to redefine segments to match the current partition of data that a thread is working on. This feature could help to avoid the problem sometimes seen with overly-aggressive checkout invalidations by making checkout operations more targeted.

Programming Language and Runtime Support. We have presented ASM a model independent of a particular programming language or runtime. It could be interesting to see how one might use ASM to build a co-designed language, though. Many recent language proposals for multithreaded codes feature thread-private memory similar to ASM's working memory [28,32,33,41,174]. We think these languages as is would map well onto ASM, and may even benefit from alterations specific to the ASM model, e.g., making checkout and checkin language primitives.

7.2.2 Using ASM Ideas in Conventional Systems

ASM-Enhanced Cache Coherence. In this dissertation we built a prototype ASM system that intentionally pushed the envelope of the ASM model. More conventional systems, such as those using traditional coherence protocols, could also benefit from ASM. ASM segments could be used to reduce directory state by serving as an indication of whether or not a particular block needs to be tracked. ASM could also be used to improve the efficiency of coherence transitions, for example by allowing an implementation to easy eliminate extraneous invalidations associated with migratory data.

Checkout and Checkin as Prefetch Hints. The semantics of checkout are similar to software prefetch instructions, and effectively tell hardware that data is about to be accessed. Hardware could exploit this by using checkout and checkin history in conjunction with prefetching units. Because checkout and checkin usually do not mean that thread is about to access an *entire* segment, such hardware would likely want to use usage information associated with the operations (such as the valid/dirty set information that is already tracked, for example, by ASM-CMP).

Mixing ASM and Conventional Coherence. We believe one promising application of the ASM model is its use in conjunction with integrated custom accelerators. In such scenarios, it is possible that an accelerator using the ASM model would need to interact with a conventional core using traditional cache coherence. Thus, an interesting future direction is to examine how ASM can interact with full blown cache coherence in the same system. In particular, we think that one would need to carefully consider what it means to check in a segment to an address space that another component thinks is coherent. The conclusions of such a study will likely be highly dependent on the particular consistency model assumed by the coherent components.

Long Pointer Propagation in Conventional Systems. There seems to be mounting support for a redesign of virtual memory in conventional systems in order to reduce the energy and performance impact of traditional paged memory. We think that long pointer propagation could be an interesting solution to the problem. In particular, long pointer propagation maintains the flat address space afforded by paged virtual memory, making the software impact minor, while eliminating expensive TLB misses. In our view, the biggest obstacle to overcome is the fragmentation problem of segments, though there is evidence that systems now have enough physical memory that fragmentation is less of a concern [118,124].

Moving to segments rather than paging could have additional benefits other than just energy and performance. The security community recently identified buffer overflows as the third biggest security hole in modern computer systems [188]. A move to segments that perform bounds checks could help to eliminate that vulnerability. Segments can also help to amortize the cost of operations performed on related data. For example, Nagarakatte, Martin, and Zdancewic recently proposed a mechanism to implement a use-after-free checker in hardware for programs written in C/C++ [140]. Their solution requires additional hardware state and memory accesses to track metadata at a fine granularity. We believe a move to segments could help to amortize this cost by integrating their checks with segments rather than individual memory accesses.

7.2.3 ASM-CMP Improvements and Optimizations

There are many opportunities to improve the ASM-CMP design to improve performance, reduce the required amount of state even further, and to provide better support for strong acoherence.

Optimizing Empty Checkout/Checkin. In our evaluation, we found that it is common for software to perform checkouts on segments that haven't been accessed and checkins on segments

that have not been modified. Currently, ASM-CMP does not try to detect these occurrences. All checkouts and checkins still go through the timestamp acquisition procedure regardless of whether or not they will have any effect. Future iterations may be able to detect these situations early an avoid the atomic updates and L2 accesses associated with the timestamp mechanism.

Non-Speculative Support for Strong Acoherence. An interesting extension of ASM-CMP, especially in terms of research value, would be support for strong acoherence. Recall that strong acoherence is difficult to support because it has nearly unbounded buffering requirements in the worst case. In the common case, however, the buffering requirements are manageable. We take advantage of this observation with best-effort acoherence, but that has the downside of increasing software complexity since it must be able to handle best-effort failures.

There has been work in the programming languages and systems communities that provide software with semantics similar to strong acoherence, usually done in the name of deterministic multiprocessing [14,32,120,174]. To support the semantics, these systems either use virtual memory protections to implement copy-on-write when concurrent updates occur, use high-level language guarantees to ensure such updates do not occur, or do so in a controlled fashion, or a combination of both. We believe that ASM-CMP could use some of these ideas to implement strong acoherence without speculation support.

For example, ASM-CMP could use something like the previous copy-on-write mechanisms to redirect blocks evicted from the L1 cache to thread-private storage (i.e., copy-on-evict rather than copy-on-write). At checkout and checkin points, the system would then need to reconcile these overflow areas with main memory.

Reducing the Overhead of Byte Diffing. The largest overhead in ASM-CMP is the metadata required to track dirty state information on a per-byte basis. Recall that this information is used

to avoid false conflicts that might arise from multiple concurrent writers of a cache block. Currently, we make no attempt to compress this state, but we believe there is plenty of opportunity to do so.

First, we observe that actual false conflicts due to byte-level sharing are rare. This would indicate that an implementation could safely forgo byte-level dirty tracking for most blocks in the cache. Second, we observe that even when there is a false conflict problem, the bitmask patterns are highly predictable for a given workload. This would suggest that a small lookup table could be used to store recently observed patterns. In cache blocks we could store an index into the lookup table rather than the full bitmask itself.

Finally, there may be an opportunity to work with a compiler to reduce the dirty bitmask state. If, for example, a compiler could guarantee that threads do not share data on anything smaller than a word, ASM-CMP could track dirty state on a per-word, rather than per-byte, granularity, thereby reducing state by 4x.

Smaller Segment Sidecars. The current ASM-CMP implementation stores a fully decoded segment descriptor in the register sidecars. If designers find this state to be excessive, they could reduce the size of the sidecar by storing a segment ID rather than a segment descriptor. The IDs would need to be decoded on every pointer dereference, requiring more Segment Lookup Table (SLT) accesses. In this case, the SLT would be used just like a TLB, i.e., on every memory access. Notably, this change would not change the size of the SLT, which is a function of the number of segments accessed in a small temporal window, and thus the SLT would still be smaller than a TLB (e.g., 8 entries vs. 64 entires).

Non-contiguous Segments. There could be a programmability benefit if ASM-CMP were to support non-contiguous segments. In particular, we think supporting strided segments could

provide a big benefit for little cost. This would allow threads to partition data in common patterns found in, for example, scatter/gather algorithms.

7.2.4 Pushing ASM Software

While this dissertation provides a solid initial analysis of the potential of an ASM system, there are many software opportunities left unexplored.

Message Passing. First, we believe that ASM would be a good model to on which to build message passing primitives. Message passing is becoming an important programming paradigm in the growing datacenter market in large part because those workloads are distributed over massive warehouse-sized computers. These machines, even though they use commodity multicores, rarely use shared memory, and when they do it is often through a message passing layer. When message passing is performed through coherent shared memory, the messages are typically implemented as objects on shared in-memory queues [31]. Thus, when a message is sent from one thread to another, the contents will undergo the same unnecessary invalidations incurred by migratory data. In ASM, on the other hand, a single message send/receive can be consolidated into a checkin/checkout pair.

Software Speculation. While we gave an example of how software might use ASM in conjunction with speculative data in Section 2.5.4, we have not yet fully evaluated its potential. We believe ASM, with its exposed thread-private storage, could enable highly efficient speculation runtimes compared to the state of the art. For example, ASM could make best-effort hybrid transactional memory runtimes comparable in performance to hardware transactional memory systems by eliminating the overhead of version management in the common case. And, compared to existing hybrid proposals that only use the store buffer for speculative state [168],

an ASM-based system could permit much larger transactions before incurring performance penalties.

7.3 Final Thoughts

To conclude this dissertation, I offer some opinions I have formed through the course of this work. I provide these opinions without supporting evidence, and present only as personal beliefs.

Multithreaded programs share a surprisingly small amount of data. In some ways this seems obvious in retrospect, but I think there is common misconception to the contrary. We have spent years optimizing coherence protocols to enable fast, fine-grained sharing, but no matter how hard we try to optimize, I don't believe we will ever overcome the fundamental problem that communication is slow and fine-grained sharing is complex.

I think there was hope that the introduction of multicore processors would change the communication tradeoffs just enough so that software could finally do the fine-grained interactions researchers had dreamed of. This has by and large been proven false. At their onset, multicores were largely used for multiprogramming (including virtual machines) where there was no sharing to speak of in applications. More recently we have seen multithreaded programs become more prevalent, but still they do most sharing in a coarse-grained fashion. Even though communication is faster is a CMP relative to an SMP, it is still not as fast as working privately.

Aside from the fundamental problem of communication latency, there is also a complexity issue with fine-grained sharing. Fine grained sharing requires fine grained synchronization. Fine grained synchronization is notoriously difficult to reason about and leads to tricky debugging situations like deadlock and Heisenbugs. By providing coherence, we give programmers the false impression that fine-grained sharing is acceptable. I believe one of the reasons that shared

memory programs are so hard to tune for performance is that programmers often do not understand when and how threads in an application communicate. Why should they? Our models are built so they can ignore it.

And so therein lies the major contribution of this work. I do not believe that we will ever reach a point where fine grained, large scale communication will be common or even desired. Yet we spend an inordinate amount of resources and effort trying to support massive fine-grained sharing in computers. Thomas Puzak once said that “Everyone knows Amdahl’s Law, but quickly forgets it.” It seems that has held true for coherence.

Bibliography

1. Abdel-Shafi, H., Hall, J., Adve, S.V., and Adve, V.S. An Evaluation of Fine-Grain Producer-Initiated Communication in Cache-Coherent Multiprocessors. *Proceedings of the Third IEEE Symposium on High-Performance Computer Architecture*, (1997).

2. Adve, S.V. and Gharachorloo, K. Shared Memory Consistency Models: A Tutorial. *IEEE Computer 29*, 12 (1996), 66–76.

3. Adve, S.V. and Hill, M.D. Weak Ordering - A New Definition. *Proceedings of the 17th Annual International Symposium on Computer Architecture*, (1990), 2–14.

4. Adve, S.V. and Hill, M.D. A Unified Formalization of Four Shared-Memory Models. *IEEE Transactions on Parallel Distributed Systems 4*, 6 (1993), 613–624.

5. Adve, S.V. Designing Memory Consistency Models for Shared-Memory Multiprocessors. 1993.

6. Agarwal, A. The Tile processor: A 64-core multicore for embedded processing. *Proceedings of HPEC Workshop*, (2007).

7. Ahamad, M., Bazzi, R.A., John, R., Kohli, P., and Neiger, 71. The Power of Processor Consistency. *Proceedings of the fifth annual ACM Symposium on Parallel Algorithms and Architectures*, (1993), 251–260.

8. Ahuja, S., Carriero, N., and Gelernter, D. Linda and Friends. *Computer 19*, 8 (1986), 26–34.

9. Akl, S.G. and Santoro, N. Optimal parallel merging and sorting without memory conflicts. *IEEE Trans. Comput. 36*, 11 (1987), 1367–1369.

10. Alameldeen, A.R. and Wood, D.A. Addressing Workload Variability in Architectural Simulations. *IEEE Micro 23*, 6 (2003), 94–98.

11. Ananian, C.S., Asanovic, K., Kuszmaul, B.C., Leiserson, C.E., and Lie, S. Unbounded Transactional Memory. *Proceedings of the Eleventh IEEE Symposium on High-Performance Computer Architecture*, (2005).

12. ARM Corporation. *ARM Architecture Reference Manual, Section A3.8*. ARM Corporation, 2007.

13. Austin, T., Larson, E., and Ernst, D. SimpleScalar: An Infrastructure for Computer System Modeling. *IEEE Computer 35*, 2 (2002), 59–67.

14. Aviram, A., Weng, S.-C., Hu, S., and Ford, B. Efficient system-enforced deterministic parallelism. *Proc. of the 9th USENIX conf. on Operating systems design and implementation*, USENIX Association (2010), 1–16.

15. Barham, P., Dragovic, B., Fraser, K., et al. Xen and the Art of Virtualization. *Proceedings of the nineteenth ACM symposium on Operating systems principles and practice (SOSP '03)*, (2003), 164–177.

16. Barroso, L.A., Gharachorloo, K., and Bugnion, E. Memory System Characterization of Commercial Workloads. *Proceedings of the 25th Annual International Symposium on Computer Architecture*, (1998), 3–14.

17. Barrow-Williams, N., Fensch, C., and Moore, S. A communication characterisation of Splash-2 and Parsec. *Workload Characterization, 2009. IISWC 2009. IEEE International Symposium on*, (2009), 86 –97.

18. Baugh, L., Neelakantam, N., and Zilles, C. Using Hardware Memory Protection to Build a High-Performance, Strongly-Atomic Hybrid Transactional Memory. *Proceedings of the 35th Annual International Symposium on Computer Architecture*, (2008).

19. Baugh, L., Neelakantam, N., and Zilles, C. Using Hardware Memory Protection to Build a High-Performance, Strongly-Atomic Hybrid Transactional Memory. *Proceedings of the 35th Annual International Symposium on Computer Architecture*, (2008).

20. Bennett, J.K., Carter, J.B., and Zwaenepoel, W. Munin: Distributed Shared Memory Based on Type-Specific Memory Coherence. *Proceedings of the 2nd ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPOPP)*, (1990).

21. Bergan, T., Anderson, O., Devietti, J., Ceze, L., and Grossman, D. CoreDet: a compiler and runtime system for deterministic multithreaded execution. *Proc. of the 15th Intnl. Conf. on Architectural Support for Programming Languages and Operating Systems*, (2010), 53–64.

22. Bergman, K., Borkar, S., Campbell, D., et al. *Exascale computing study: Technology challenges in achieving exascale systems*. DARPA Technical Report, 2008.

23. Bershad, B.N., Zekauskas, M.J., and Sawdon, W.A. The Midway Distributed Shared Memory System. *Compcon Spring '93, Digest of Papers*, (1993), 528–537.

24. Binkert, N., Beckmann, B., Black, G., et al. The gem5 simulator. *Computer Architecture News (CAN)*, (2011).

25. Binkert, N.L., Dreslinski, R.G., Hsu, L.R., Lim, K.T., Saidi, A.G., and Reinhardt, S.K. The M5 Simulator: Modeling Networked Systems. *IEEE Micro 26*, (2006), 52–60.

26. Blumofe, R.D., Joerg, C.F., Kuszmaul, B.C., Leiserson, C.E., Randall, K.H., and Zhou, Y. Cilk: an efficient multithreaded runtime system. *Proc. of the 4th ACM SIGPLAN Symp. on Principles and Practice of Parallel Programming (PPOPP)*, (1991), 207–216.

27. Bobba, J., Moore, K.E., Volos, H., et al. Performance Pathologies in Hardware Transactional Memory. *Proceedings of the 34th Annual International Symposium on Computer Architecture*, (2007).

28. Bocchino,Jr., R.L., Adve, V.S., Dig, D., et al. A type and effect system for deterministic parallel Java. *Proceedings of the 24th ACM SIGPLAN conference on Object oriented programming systems languages and applications*, ACM (2009), 97–116.

29. Boehm, H.-J. and Adve, S.V. Foundations of the C++ Concurrency Memory Model. *PLDI 2008*, (2008), 68–78.

30. Brooks, D., Tiwari, V., and Martonosi, M. Wattch: A Framework for Architectural-Level Power Analysis and Optimizations. *Proceedings of the 27th Annual International Symposium on Computer Architecture*, (2000), 83–94.

31. Buntinas, D., Mercier, G., and Gropp, W. Design and Evaultation of Nemesis, a scalable, low-latency, message-passing communication subsystem. *Cluster Computing and the Grid (CCGRID)*, (2006).

32. Burckhardt, S., Baldassin, A., and Leijen, D. Concurrent Programming with Revisions and Isolation Types. *Proceedings of the ACM international conference on Object oriented programming systems languages and applications*, ACM (2010), 691–707.

33. Carlson, W.W., Draper, J.M., Culler, D.E., Yelick, K., Brooks, E., and Warren, K. *Introduction to UPC and language specification*. Center for Computing Sciences, Institute for Defense Analyses, 1999.

34. Carter, J.B., Bennett, J.K., and Zwaenepoel, W. Implementation and Performance of Munin. *Proceedings of the 13th ACM Symposium on Operating Systems Principles*, (1991).

35. Carter, N.P., Keckler, S.W., and Dally, W.J. Hardware support for fast capability-based addressing. *Proceedings of the sixth international conference on Architectural support for programming languages and operating systems*, ACM (1994), 319–327.

36. Cavé, V., Zhao, J., Shirako, J., and Sarkar, V. Habanero-Java: the New Adventures of Old X10. *Proceedings of the 9th International Conference on the Principles and Practice of Programming in Java (PPPJ)*, (2011).

37. Cederqvist, P. *Version Management with CVS*. Free Software Foundation, Inc., 1993.

38. Chakraborty, K., Wells, P., and Sohi, G. *A Case for Over-provisioned Multicore System*. University of Wisconsin Technical Report, 2007.

39. Chamberlain, B.L., Callahan, D., and Zima, H.P. Parallel Programmability and the Chapel Language. *International Journal of High Performance Computing Applications*, (2007).

40. Chandra, S., Richards, B., and Larus, J.R. Teapot: Language Support for Writing Memory Coherence Protocols. *Proceedings of the SIGPLAN 1996 Conference on Programming Language Design and Implementation*, (1996).

41. Charles, P., Donawa, C., Ebcioglu, K., et al. X10: An Object-Oriented Approachto Non-Uniform Cluster Computing. *Object-Oriented Programming, Systems, Languages and Applications*, (2005).

42. Choi, B., Komuravelli, R., Sung, H., et al. Denovo: Rethinking the Memory Hierarchy for Disciplined Parallelism. *20th International Conf. on Parallel Architectures and Compilation Techniques (PACT)*, (2011).

43. Chuang, W., Narayanasmy, S., Venkatesh, G., et al. Unbounded Page-Based Transactional Memory. *Proceedings of the Twelfth International Conference on Architectural Support for Programming Languages and Operating Systems*, (2006).

44. Chung, E.S., Milder, P.A., Hoe, J.C., and Mai, K. Single-Chip Heterogeneous Computing: Does the Future Include Custom Logic, FPGAs, and GPGPUs? *Proceedings of the 2010 43rd Annual IEEE/ACM International Symposium on Microarchitecture*, IEEE Computer Society (2010), 225–236.

45. Corbató, F.J. and Vyssotsky, V.A. Introduction and overview of the Multics system. *Proceedings of the November 30–December 1, 1965, fall joint computer conference, part I*, (1965), 185–196.

46. Corella, F., Stone, J.M., and Barton, C.M. *A Formal Specification of the PowerPC Shared Memory Architecture*. IBM, 1993.

47. Culler, D.E., Dusseau, A., Goldstein, S.C., et al. Parallel programming in Split-C. *Supercomputing'93. Proceedings*, (1993), 262–273.

48. Culler, D.E. and Singh, J.P. *Parallel Computer Architecture: A Hardware/Software Approach*. Morgan Kaufmann Publishers, Inc., 1999.

49. Dally, W.J. and Towles, B.P. *Principles and Practices of Interconnection Networks*. Morgan Kaufmann, 2003.

50. Damron, P., Fedorova, A., Lev, Y., Luchango, V., Moir, M., and Nussbaum, D. Hybrid Transactional Memory. *Proceedings of the Twelfth International Conference on Architectural Support for Programming Languages and Operating Systems*, (2006).

51. Dennard, R.H., Gaensslen, F.H., Rideout, V.. L., Bassous, E., and LeBlanc, A.R. Design of ion-implanted MOSFET's with very small physical dimensions. *IEEE Journal of Solid-State Circuits 9*, 5 (1974), 256–268.

52. DeOrio, A., Bauserman, A., and Bertacco, V. Post-silicon verification for cache coherence. *Computer Design, 2008. ICCD 2008. IEEE International Conference on*, (2008), 348–355.

53. Devietti, J., Lucia, B., Ceze, L., and Oskin, M. DMP: Deterministic Shared Memory Multiprocessing. *Proceedings of the Fourteenth International Conference on Architectural Support for Programming Languages and Operating Systems*, (2009), 85–96.

54. Dijkstra, E.W. Solution of a problem in concurrent programming control. *Commun. ACM 8*, 9 (1965), 569.

55. Dill, D.L., Drexler, A.J., Hu, A.J., and Yang, C.H. Protocol Verification as a Hardware Design Aid. *International Conference on Computer Design*, (1992).

56. Dubois, M., Scheurich, C., and Briggs, F. Memory Access Buffering in Multiprocessors. *Proceedings of the 13th annual International Symposium on Computer Architecture*, (1986), 434–442.

57. Epperson, J.F. *An Introduction to Numerical Methods and Analysis*. Wiley-Interscience, 2007.

58. Esmaeilzadeh, H., Blem, E., St. Amant, R., Sankaralingam, K., and Burger, D. Dark silicon and the end of multicore scaling. *Proceedings of the 38th Annual International Symposium on Computer Architecture 39*, 3 (2011), 365–376.

59. Feng, M. and Leiserson, C.E. Efficient detection of determinacy races in Cilk programs. *Proceedings of the ninth annual ACM symposium on Parallel algorithms and architectures*, ACM (1997), 1–11.

60. Fensch, C. and Cintra, M. An OS-based alternative to full hardware coherence on tiled CMPs. *High Performance Computer Architecture, 2008. HPCA 2008. IEEE 14th International Symposium on*, (2008), 355 –366.

61. Ferdman, M., Adileh, A., Kocberber, O., et al. Clearing the Clouds: A Study of Emerging Scale-out Workloads on Modern Hardware. *Proceedings of the 17th Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS 2012)*, ACM (2012).

62. Ferdman, M., Lotfi-Kamran, P., Balet, K., and Falsafi, B. Cuckoo directory: A scalable directory for many-core systems. *Proc. of the 17th IEEE Symp. on High-Performance Computer Architecture*, (2011).

63. Flanagan, D. and Matsumoto, Y. *The ruby programming language*. O'Reilly Media, 2008.

64. Freescale Semiconductor. *MSC8144 Reference Manual*. 2009.

65. Frigo, M., Halpern, P., Leiserson, C.E., and Lewin-Berlin, S. Reducers and other Cilk++ hyperobjects. *Proceedings of the twenty-first annual symposium on Parallelism in algorithms and architectures*, (2009), 79–90.

66. Frigo, M., Leiserson, C.E., Prokop, H., and Ramachandran, S. Cache-oblivious algorithms. *Foundations of Computer Science, 1999. 40th Annual Symposium on*, (1999), 285 –297.

67. Frigo, M., Leiserson, C.E., and Randall, K.H. The implementation of the Cilk-5 multithreaded language. *ACM Sigplan Notices 33*, 5 (1998), 212–223.

68. Frigo, M. and Strumpen, V. Cache oblivious stencil computations. *Proceedings of the 19th annual international conference on Supercomputing*, ACM (2005), 361–366.

69. Gay, D. and Aiken, A. Memory Management with Explicit Regions. *Proceedings of the 1998 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, (1998).

70. Gharachorloo, K., Lenoski, D., Laudon, J., Gibbons, P., Gupta, A., and Hennessy, J. Memory Consistency and Event Ordering in Scalable Shared-Memory. *Proc. of the 17th International Symp. on Computer Architecture*, (1990), 15–26.

71. Gibson, J.S. Memory profiling on shared-memory multiprocessors. 2002.

72. Gil, J.R.T., Sanchez, M.E.A., and Carrasco, J.M.G. Characterization of Conflicts in Log-Based Transactional Memory (LogTM). *Proceedings of the 16th EUROMICRO Conference on Parallel, Distributed and Network-Based Processing (PDP)*, (2008).

73. Gniady, C., Falsafi, B., and Vijaykumar, T.N. Is SC + ILP = RC? *Proceedings of the 26th Annual International Symposium on Computer Architecture*, (1999), 162–171.

74. González, A., Aliagas, C., and Valero, M. A data cache with multiple caching strategies tuned to different types of locality. *Proceedings of the 9th international conference on Supercomputing*, ACM (1995), 338–347.

75. Gray, J., Lorie, R., Putzolu, F., and Traiger, I. Granularity of Locks and Degrees of Consistency in a Shared Database. *Modeling in Data Base Management Systems, Elsevier North Holland, New York*, (1975).

76. Gray, J. and Reuter, A. *Transaction Processing: Concepts and Techniques*. Morgan Kaufmann, 1993.

77. Gropp, W. and Lusk, E. Goals Guiding Design: PVM and MPI. *Proceedings of the IEEE International Conference on Cluster Computing*, IEEE Computer Society (2002), 257–.

78. Guo, Y., Zhao, J., Cave, V., and Sarkar, V. SLAW: a scalable locality-aware adaptive work-stealing scheduler. *Parallel & Distributed Processing (IPDPS 2010)*, (2010), 1–12.

79. Gupta, A. and Weber, W.-D. Cache Invalidation Patterns in Shared-Memory Multiprocessors. *IEEE Transactions on Computers 41*, 7 (1992), 794–810.

80. Gustavson, D. The Scalable Coherent Interface and related standards projects. *IEEE Micro 12*, 1 (1992), 10–22.

81. Hammond, L., Wong, V., Chen, M., et al. Transactional Memory Coherence and Consistency. *Proceedings of the 31st Annual International Symposium on Computer Architecture*, (2004).

82. Harris, T., Larus, J.R., and Rajwar, R. *Transactional Memory, 2nd Edition*. Morgan & Claypool Publishers, 2010.

83. Held, J. Single-chip Cloud Computer: An experimental many-core processor from Intel Labs. 2010. http://communities.intel.com/servlet/JiveServlet/downloadBody/5074-102-1-8131/SCC_Sympossium_Feb212010_FINAL-A.pdf.

84. Hennessy, J.L. and Patterson, D.A. *Computer Architecture: A Quantitative Approach, 4th Edition*. Morgan Kaufmann, 2006.

85. Herlihy, M. and Moss, J.E.B. *Transactional Memory: Architectural Support for Lock-Free Data Structures*. Digital Cambridge Research Lab, 1992.

86. Hill, M.D., Hower, D., Moore, K.E., Swift, M.M., Volos, H., and Wood, D.A. *A Case for Deconstructing Hardware Transactional Memory Systems*. University of Wisconsin-Madison, 2007.

87. Hill, M.D. and Marty, M.R. Amdahl's Law in the Multicore Era. *IEEE Computer*, (2008), 33–38.

88. Hill, M.D. Multiprocessors Should Support Simple Memory Consistency Models. *IEEE Computer 31*, 8 (1998), 28–34.

89. Horowitz, M., Alon, E., Naffziger, S., Kumar, R., and Bernstein, K. Scaling, power, and the future of CMOS. *IEEE International Electron Devices Meeting, 2005.*, (2005).

90. Hower, D.R., Dudnik, P., Wood, D.A., and Hill, M.D. Calvin: Deterministic or Not? Free Will to Choose. *Proc. of the 17th IEEE Symp. on High-Performance Computer Architecture*, (2011).

91. Hruska, J. Nvidia deeply unhappy with TSMC, claims 20nm essentially worthless | ExtremeTech. *ExtremeTech*. http://www.extremetech.com/computing/123529-nvidia-deeply-unhappy-with-tsmc-claims-22nm-essentially-worthless.

92. IEEE Portable Applications Standards Committee. *IEEE Std 1003.1 c-1995, Threads Extensions*. IEEE, 1995.

93. Intel Corporation. *Intel 64 and IA-32 Architectures Software Developer's Manual, Volume 3A, Part 1*. 2009.

94. Intel Corporation. *Intel 64 and IA-32 Architectures Software Developer's Manual*. Intel Corporation.

95. Johnson, D.R., Johnson, M.R., Kelm, J.H., Tuohy, W., Lumetta, S.S., and Patel, S.J. Rigel: A 1024-Core Single-Chip Accelerator Architecture. *IEEE Micro*, (2011).

96. Johnson, K.L., Kaashoek, M.F., and Wallach, D.A. CRL: high-performance all-software distributed shared memory. *SIGOPS Oper. Syst. Rev. 29*, 5 (1995), 213–226.

97. Johnson, T.L. and Hwu, W.W. Run-time adaptive cache hierarchy management via reference analysis. *SIGARCH Comput. Archit. News 25*, 2 (1997), 315–326.

98. Kahle, J.A., Day, M.N., Hofstee, H.P., Johns, C.R., Maeurer, T.R., and Shippy, 68. Introduction to the cell multiprocessor. *IBM J. Res. Dev. 49*, 4/5 (2005), 589–604.

99. Karam, L.J., AlKamal, I., Gatherer, A., Frantz, G.A., Anderson, D.V., and Evans, B.L. Tends in Multi-Core DSP Platforms. *Signal Processing Magazine, IEEE 26*, 6 (2009), 38–49.

100. Keckler, S.W., Dally, W.J., Khailany, B., Garland, M., and Glasco, D. GPUs and the Future of Parallel Computing. *Micro, IEEE 31*, 5 (2011), 7 –17.

101. Keleher, P., Cox, A.L., and Zwaenapoel, W. Lazy Release Consistency for Software Distributed Shared Memory. *Proc. of the 19th International Symp. on Computer Architecture*, (1992), 13–21.

102. Keleher, P., Dwarkadas, S., Cox, A., and Zwaenepoel, W. *TreadMarks: Distributed Shared Memory on Standard Workstations and Operating Systems*. Rice University, 1993.

103. Kelm, J.H., Johnson, D.R., Tuohy, W., Lumetta, S.S., and Patel, S.J. Cohesion: A Hybrid Memory Model for Accelerators. *Proccedings of the 37th Annual Internal Symposium on Computer Architecture (ISCA)*, (2010).

104. Knauerhase, R., Cledat, R., and Teller, J. For Extreme Parallelism, Your OS Is Sooooo Last-Millennium. *HotPar '12*, (2012).

105. Kroft, D. Lockup-Free Instruction Fetch/Prefetch Cache Organization. *Proc. 8th Symposium on Computer Architecture, Computer Architecture News*, (1981), 81–87.

106. Kumar, S., Chu, M., Hughes, C.J., Kundu, P., and Nguyen, A. Hybrid Transactional Memory. *Proceedings of the 13th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP)*, (2006), 209–220.

107. Lamport, L. Time, Clocks and the Ordering of Events in a Distributed System. *Communications of the ACM 21*, 7 (1978), 558–565.

108. Lamport, L. How to Make a Multiprocessor Computer that Correctly Executes Multiprocess Programs. *IEEE Transactions on Computers C-28*, 9 (1979), 690–691.

109. Lamport, L. *How to Write a Proof*. Digital Equipment Corp., 1993.

110. Larus, G.H.J., Abadi, M., Aiken, M., et al. *An Overview of the Singularity Project*. Microsoft Research, 2005.

111. Larus, J. and Hunt, G. The Singularity system. *Commun. ACM 53*, 8 (2010), 72–79.

112. Laudon, J. and Lenoski, D. The SGI Origin: A ccNUMA Highly Scalable Server. *Proceedings of the 24th Annual International Symposium on Computer Architecture*, (1997), 241–251.

113. Lebeck, A.R. and Wood, D.A. Dynamic Self-Invalidation: Reducing Coherence Overhead in Shared-Memory Multiprocessors. *Proceedings of the 22nd Annual International Symposium on Computer Architecture*, (1995), 48–59.

114. Lee, H.-H.S. and Tyson, G.S. Region-based caching: an energy-delay efficient memory architecture for embedded processors. *Proceedings of the 2000 international conference on Compilers, architecture, and synthesis for embedded systems*, ACM (2000), 120–127.

115. Leijen, D., Schulte, W., and Burckhardt, S. The design of a task parallel library. *Proceedings of the 24th ACM SIGPLAN conference on Object oriented programming systems languages and applications*, ACM (2009), 227–242.

116. Lenoski, D., Laudon, J., Gharachorloo, K., et al. The Stanford DASH Multiprocessor. *IEEE Computer 25*, 3 (1992), 63–79.

117. Lev, Y., Moir, M., and Nussbaum, D. PhTM: Phased Transactional Memory. *Proceedings of the Second ACM SIGPLAN Workshop on Languages, Compilers, and Hardware Support for Transactional Computing*, (2007).

118. Levin, J. *Mac OS X and iOS Internals: To the Apple's Core*. Wrox, 2012.

119. Li, S., Ahn, J.H., Strong, R.D., Brockman, J.B., Tullsen, D.M., and Jouppi, N.P. McPAT: an integrated power, area, and timing modeling framework for multicore and manycore architectures. *MICRO 42*, ACM (2009), 469–480.

120. Liu, T., Curtsinger, C., and Berger, E. Dthreads: Efficient and Deterministic Multithreading. *ACM Symposium and Operating Systems Principles (SOSP)*, (2011).

121. Lupon, M., Magklis, G., and Gonzalez, A. FASTM: A Log-based Hardware Transactional Memory with Fast Abort Recovery. *Proceedings of the 18th International Conference on Parallel Architectures and Compilation Techniques*, (2009).

122. Manson, J., Pugh, W., and Adve, S.V. The Java Memory Model. *POPL '05: Proc. of the 32nd Symposium on Principles of Programming Languages*, (2005), 378–391.

123. Marathe, V.J., III, W.N.S., and Scott, M.L. Adaptive Software Transactional Memory. *Distributed algorithms*, Springer-Verlag GmbH (2005), 354–368.

124. Mars, J., Tang, L., Hundt, R., Skadron, K., and Soffa, M.L. Bubble-Up: increasing utilization in modern warehouse scale computers via sensible co-locations. *Proc. of the 44th Annual IEEE/ACM International Symp. on Microarchitecture*, (2011).

125. Martin, M.M.K., Sorin, D.J., Beckmann, B.M., et al. Multifacet's General Execution-driven Multiprocessor Simulator (GEMS) Toolset. *Computer Architecture News*, (2005), 92–99.

126. Martin, M.M.K. Token Coherence. 2003.

127. May, I.B.M. IBM System/370 Principles of Operation. *Publication Number GA22-7000-9, File*, S370-01 (1983).

128. Mellor-Crummey, J.M. and Scott, M.L. Algorithms for Scalable Synchronization on Shared-Memory Multiprocessors. *ACM Transactions on Computer Systems 9*, 1 (1991), 21–65.

129. Min, S.J., Iancu, C., and Yelick, K. Hierarchical work stealing on manycore clusters. *Fifth Conference on Partitioned Global Address Space Programming Models (PGAS11)*, (2011).

130. Minh, C.C., Trautmann, M., Chung, J., et al. An Effective Hybrid Transactional Memory System with Strong Isolation Guarantees. *Proceedings of the 34th Annual International Symposium on Computer Architecture*, (2007).

131. MIPS Technologies. *MIPS Architecture For Programmers Volume I-A: Introduction to the MIPS32 Architecture*. 2011.

132. MIPS Technologies. *MIPS Architecture For Programmers Volume II: The MIPS32 Instruction Set*. 2011.

133. MIPS Technologies. *MIPS Architecture For Programmers Volume III: The MIPS32 and microMIPS32 Privileged Resource Architecture*. 2011.

134. Monnerat, L.R. and Bianchini, R. Efficiently adapting to sharing patterns in software DSMs. *High-Performance Computer Architecture, 1998. Proceedings., 1998 Fourth International Symposium on*, (1998), 289 –299.

135. Moore, G.E. Cramming More Components onto Integrated Circuits. *Electronics*, (1965), 114–117.

136. Moore, G.E. Progress in digital integrated electronics. *Electron Devices Meeting, 1975 International*, (1975), 11 – 13.

137. Moore, K.E., Bobba, J., Moravan, M.J., Hill, M.D., and Wood, D.A. LogTM: Log-Based Transactional Memory. *Proceedings of the Twelfth IEEE Symposium on High-Performance Computer Architecture*, (2006), 258–269.

138. Muralimanohar, N., Balasubramonian, R., and Jouppi, N. Optimizing NUCA Organizations and Wiring Alternatives for Large Caches with CACTI 6.0. *Proceedings of the 40th Annual IEEE/ACM International Symposium on Microarchitecture*, IEEE Computer Society (2007), 3–14.

139. Muralimanohar, N., Balasubramonian, R., and Jouppi, N.P. *CACTI 6.0*. Hewlett Packard Labs, 2009.

140. Nagarakatte, S., Martin, M.M.K., and Zdancewic, S. Watchdog: Hardware for Safe and Secure Manual Memory Management and Full Memory Safety. *Proceedings of the 39th International Symposium on Computer Architecture (ISCA 2012)*, (2012).

141. Netzer, R.H.B. and Miller, B.P. What are Race Conditions?: Some Issues and Formalizations. *ACM Letters on Programming Languages and Systems 1*, 1 (1992), 74–88.

142. Olivier, S., Huan, J., Liu, J., et al. UTS: an unbalanced tree search benchmark. *Proceedings of the 19th international conference on Languages and compilers for parallel computing*, Springer-Verlag (2007), 235–250.

143. Olszewski, M., Ansel, J., and Amarasinghe, S. Kendo: Efficient Deterministic Multithreading in Software. *Proceedings of the Fourteenth International Conference on Architectural Support for Programming Languages and Operating Systems*, (2009).

144. Olszewski, M., Cutler, J., and Steffan, J.G. JudoSTM: A Dynamic Binary-Rewriting Approach to Software Transactional Memory. *Proceedings of the International Conference on Parallel Architectures and Compilation Techniques*, (2007).

145. Olukotun, K. and Hammond, L. The Future of Microprocessors. *ACM Queue 3*, 7 (2005), 26–29.

146. Pagh, R. and Rodler, F.F. Cuckoo Hashing. *Proceedings of the 9th Annual European Symposium on Algorithms*, (2001), 121–133.

147. Project, F. *Fortress Project Home*. .

148. Rajwar, R., Herlihy, M., and Lai, K. Virtualizing Transactional Memory. *Proceedings of the 32nd Annual International Symposium on Computer Architecture*, (2005).

149. Ramachandran, U. and Khalidi, M.Y.A. An Implementation of Distributed Shared Memory. *Software - Principles and Experience 21*, 5 (1991), 443–464.

150. Randall, K.H. Cilk: Efficient Multithreaded Computing. 1998. http://supertech.csail.mit.edu/papers/randall-phdthesis.pdf.

151. Reinders, J. *Intel Threading Building Blocks: Outfitting C++ for Multi-core Processor Parallelism*. O'Reilly Media, 2007.

152. Ros, A., Acacio, M.E., and Garcia, J.M. DiCo-CMP: Efficient cache coherency in tiled CMP architectures. *Proceedings of the International Parallel and Distributed Processing Symposium Symposium*, (2008), 1–11.

153. Saha, B., Adl-Tabatabai, A.-R., Hudson, R.L., Minh, C.C., and Hertzberg, B. McRT-STM: a High Performance Software Transactional Memory System for a Multi-Core Runtime. *Proceedings of the 13th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP)*, (2006).

154. Salcianu, A. and Rinard, M. Pointer and Escape Analysis for Multithreaded Programs. *Proceedings of the 8th Symposium on Principles and Practices of Parallel Programming*, (2011), 12–23.

155. Sewell, P., Sarkar, S., Owens, S., Nardelli, F.Z., and Myreen, M.O. x86-TSO: a rigorous and usable programmer's model for x86 multiprocessors. *Commun. ACM 53*, 7 (2010), 89–97.

156. Shavit, N. and Touitou, D. Software Transactional Memory. *Fourteenth ACM Symposium on Principles of Distributed Computing, Ottawa, Ontario, Canada*, (1995), 204–213.

157. Shriraman, A., Marathe, V.J., Dwarkadas, S., et al. Hardware Acceleration of Software Transactional Memory. *Proceedings of the First ACM SIGPLAN Workshop on Languages, Compilers, and Hardware Support for Transactional Computing*, (2006).

158. Singh, J.P., Weber, W.-D., and Gupta, A. SPLASH: Stanford Parallel Applications for Shared Memory. *Computer Architecture News 20*, 1 (1992), 5–44.

159. Sites, R.L., ed. *Alpha Architecture Reference Manual*. Digital Press, 1992.

160. Sorin, D.J., Hill, M.D., and Wood, D.A. *A Primer on Memory Consistency and Cache Coherence*. Synthesis Lectures in Computer Architecture, 2011.

161. Stackhouse, B., Bhimji, S., Bostak, C., et al. A 65 nm 2-billion transistor quad-core itanium processor. *Solid-State Circuits, IEEE Journal of 44*, 1 (2009), 18–31.

162. Stenstrom, P., Brorsson, M., and Sandberg, L. Adaptive Cache Coherence Protocol Optimized for Migratory Sharing. *Proceedings of the 20th Annual International Symposium on Computer Architecture*, (1993), 109–118.

163. Sutter, H. The Free Lunch Is Over: A Fundamental Turn Toward Concurrency in Software. *Dr. Dobb's Journal 30*, 3 (2005).

164. Tarjan, D. and Skadron, K. The Sharing Tracker: Using Ideas from Cache Coherence Hardware to Reduce Off-Chip Memory Traffic with Non-Coherent Caches. *Proceedings of the 2010 ACM/IEEE International Conference for High Performance Computing, Networking, Storage, and Analysis (SC '10)*, (2010).

165. Texas Instruments. *TMS320C64x+ DSP Megamodule Reference Guide*. 2010.

166. Tool Interface Standards Committee. *Executable and Linking Format (ELF) specification*. May, 1995.

167. Torrellas, J., Lam, M.S., and Hennessy, J.L. False Sharing and Spatial Locality in Multiprocessor Caches. *IEEE Transactions on Computers 43*, 6 (1994), 651–663.

168. Tremblay, M. Transactional memory for a modern microprocessor. *Twenty-Sixth ACM Symposium on Principles of Distributed Computing*, (2007).

169. Tuck, J., Ahn, W., Ceze, L., and Torrellas, J. SoftSig: Software-Exposed Hardware Signatures for Code Analysis and Optimization. *Proceedings of the Thirteenth International Conference on Architectural Support for Programming Languages and Operating Systems*, (2008).

170. Valiant, L.G. A Bridging Model for Parallel Computation. *Communications of the ACM 33*, 8 (1990), 103–111.

171. Vangal, S.$1, Howard, 74, Ruhl, 71, et al. An 80-Tile Sub-100-W TeraFLOPS Processor in 65-nm CMOS. *IEEE Journal of Solid-State Circuits 43*, 1 (2008), 29–41.

172. Waingold, E. and others, others. Baring It All to Software: Raw Machines. *IEEE Computer*, (1997), 86–93.

173. Wang, H.-S., Zhu, X., Peh, L.-S., and Malik, S. Orion: A Power-Performance Simulator for Interconnection Networks. *Proc. of the 35th International Symp. on Microarchitecture*, (2002), 294–305.

174. Warth, A., Ohshima, Y., Kaehler, 84, and Kay, A. Worlds: controlling the scope of side effects. *Proceedings of the 25th European conference on Object-oriented programming*, Springer-Verlag (2011), 179–203.

175. Watanabe, Y., Davis, J.D., and Wood, D.A. WiDGET: Wisconsin Decoupled Grid Execution Tiles. *Proceedings of the 37th Annual International Symposium on Computer Architecture*, (2010).

176. Weaver, D.L. and Germond, T., eds. *SPARC Architecture Manual (Version 9)*. PTR Prentice Hall, 1994.

177. Wenisch, T.F., Wunderlich, R.E., Ferdman, M., Ailamaki, A., Falsafi, B., and Hoe, J.C. SimFlex: Statistical Sampling of Computer System Simulation. *IEEE Micro 26*, 4 (2006), 18–31.

178. Witchel, E., Cates, J., and Asanovic, K. Mondrian memory protection. *Proceedings of the Tenth International Conference on Architectural Support for Programming Languages and Operating Systems*, (2002), 304–316.

179. Wohltez, K. KWare HEAT3D. http://geodynamics.lanl.gov/Wohletz/Heat.htm.

180. Woo, S.C., Ohara, M., Torrie, E., Singh, J.P., and Gupta, A. The SPLASH-2 Programs: Characterization and Methodological Considerations. *Proc. of the 22nd International Symp. on Computer Architecture*, (1995), 24–37.

181. Yeager, K.C. The MIPS R10000 Superscalar Microprocessor. *IEEE Micro 16*, 2 (1996), 28–40.

182. Yuanyuan Zhou, L.I. and Li, K. Performance Evaluation of Two Home-Based Lazy Release Consistency Protocols for Shared Virtual Memory Systems. *Proceedings of the Second USENIX Symposium on Operating Systems Design and Implementation*, (1996), 75–88.

183. Zebchuk, J., Qureshi, M.K., Srinivasan, V., and Moshovos, A. A Tagless Coherence Directory. *MICRO 42*, (2009), 423–434.

184. Zhang, M., Lebeck, A.R., and Sorin, D.J. Fractal Coherence: Scalably Verifiable Cache Coherence. *Proceedings of the 2010 43rd Annual IEEE/ACM International Symposium on Microarchitecture*, IEEE Computer Society (2010), 471–482.

185. Zhao, H., Shriraman, A., Dwarkadas, S., and Srinivasan, V. SPATL: Honey, I Shrunk the Coherence Directory. *Parallel Architectures and Compilation Techniques (PACT), 2011 International Conference on*, (2011), 33 –44.

186. Zhao, L., Iyer, R., Makineni, S., Newell, D., and Cheng, L. NCID: a non-inclusive cache, inclusive directory architecture for flexible and efficient cache hierarchies. *CF '10: Proceedings of the 7th ACM international conference on Computing frontiers*, (2010).

187. *IEEE Standard for Scalable Coherent Interface (SCI)*. 1993.

188. *2011 CWE/SANS Top 25 Most Dangerous Software Errors*. CWE/SANS, 2011.

189. *21st Century Computer Architecture*. Computing Community Consortium, 2012.

190. Apache Subversion. http://subversion.apache.org/.

191. pcc portable c compiler. http://pcc.ludd.ltu.se.

192. GLIBC, the GNU C Library. http://www.gnu.org/software/libc/.

193. The Newlib Homepage. http://sourceware.org/newlib/.

194. Netlib Repository. http://www.netlib.org/.

195. The Cilk Project. *The Cilk Project*. http://supertech.csail.mit.edu/cilk/.

Appendix A. Justification of Common ASM Constraints

Here we justify the existence of each of the common memory constraints in Figure 3-5 by showing what behavior they prevent. All the examples assume weak acoherence, but the same principles apply to all segment types.

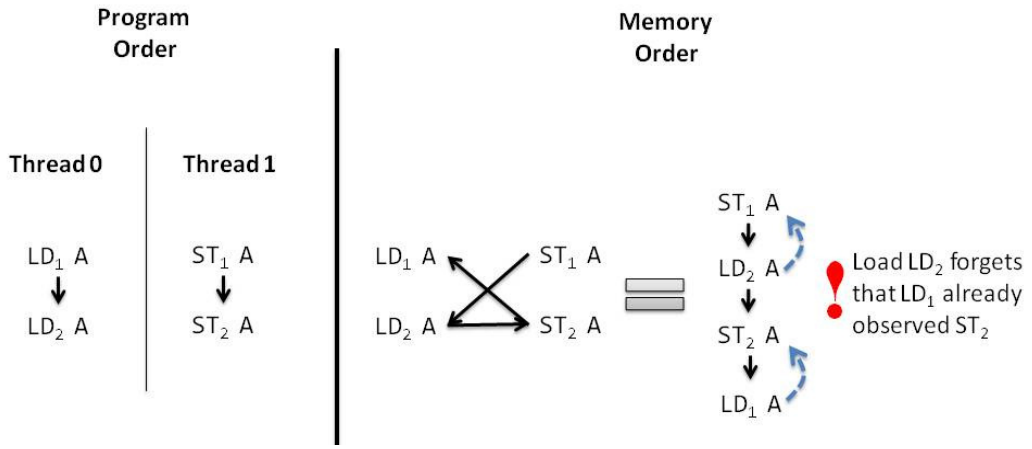

Figure A-1. Justification for common memory order constraint (3-11). The constraint prevents the execution on the right.

In Figure A-1 we show that without constraint (3-11) a program could "forget" the value of a previously observed store, leading to a lack of causality that would make writing correct programs mind bending at best. We display a possible memory order without constraint (3-11) in two visual styles for convenience. First, on the left, we draw memory order arrows with operations still in program order position to emphasize a cycle that occurs. Second, on the right, we draw memory order linearly from top to bottom to show succinctly the values loads observe, shown as blue dashed arrows. As the figure shows, without the constraint that program order follows memory order for loads to the same address, the second load L_2 on thread 0 can "forget" that an earlier store already observed the second store ST_2 on thread 1.

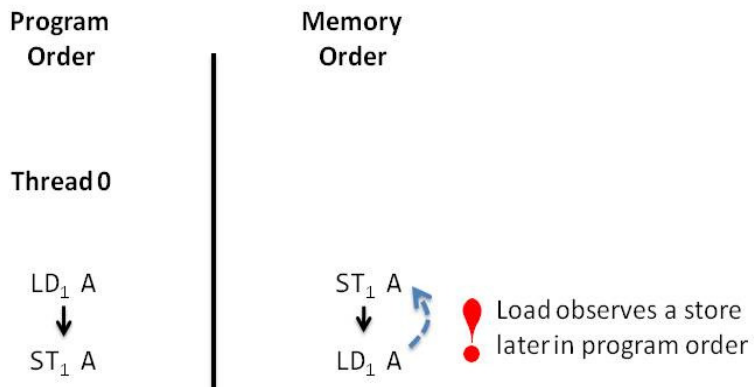

Figure A-2. Justification for common memory order constraint (3-12). The constraint prevents the execution on the right.

Without constraint (3-12), it would be possible for a load to read its own store that occurs later in program order, which we show in Figure A-2, Because in the absence of the constraint a load can be reordered with a store to the same address, load LD_1 can occur after store ST_1. By the rules of the load value equation, LD_1 could then observe the value produced by ST_1 even though it hasn't happened yet.

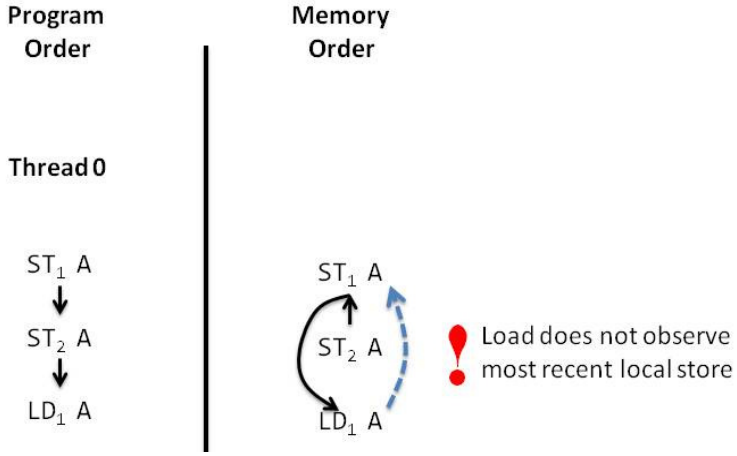

Figure A-3. Justification for common memory order constraint (3-13). The constraint prevents the execution on the right.

Constraint (3-13) ensures that store causality in program order is always maintained. Without it, as shown in Figure A-3, a thread could bypass the value of its most recent store in program order in favor a store further away in program order.

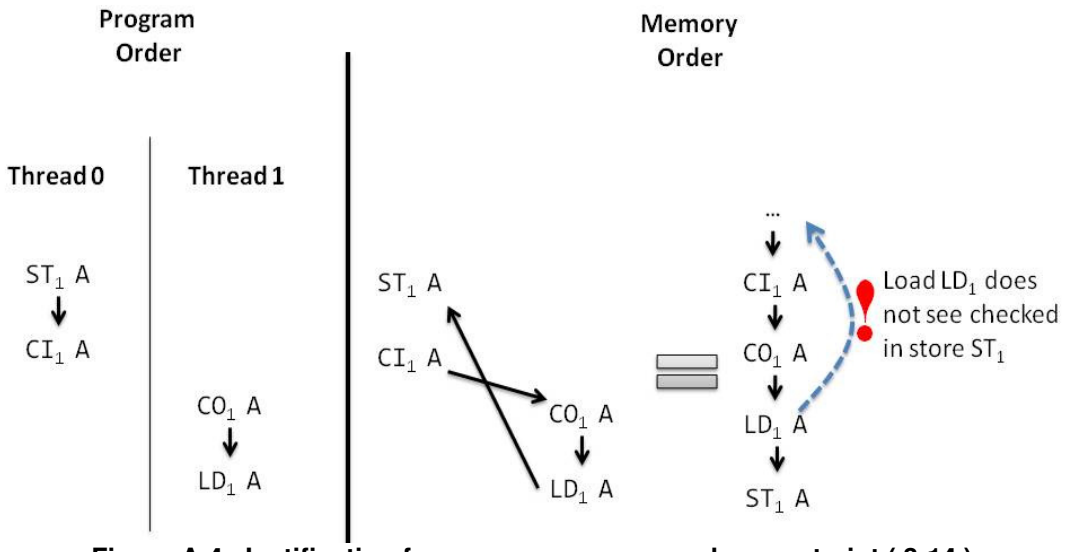

Figure A-4. Justification for common memory order constraint (3-14). The constraint prevents the execution on the right.

Constraint (3-14) ensures that after a checkout, all previously checked in stores are visible. Figure A-4 shows what could happen if constraint (3-14) were not enforced. Even though the load LD_1 occurs after a checkout ordered after the checkin that made ST_1 visible, without constraint (3-14) the load might be ordered before the store in memory order. By the load value equation, we see that if that were to happen load LD_1 would not see store ST_1.

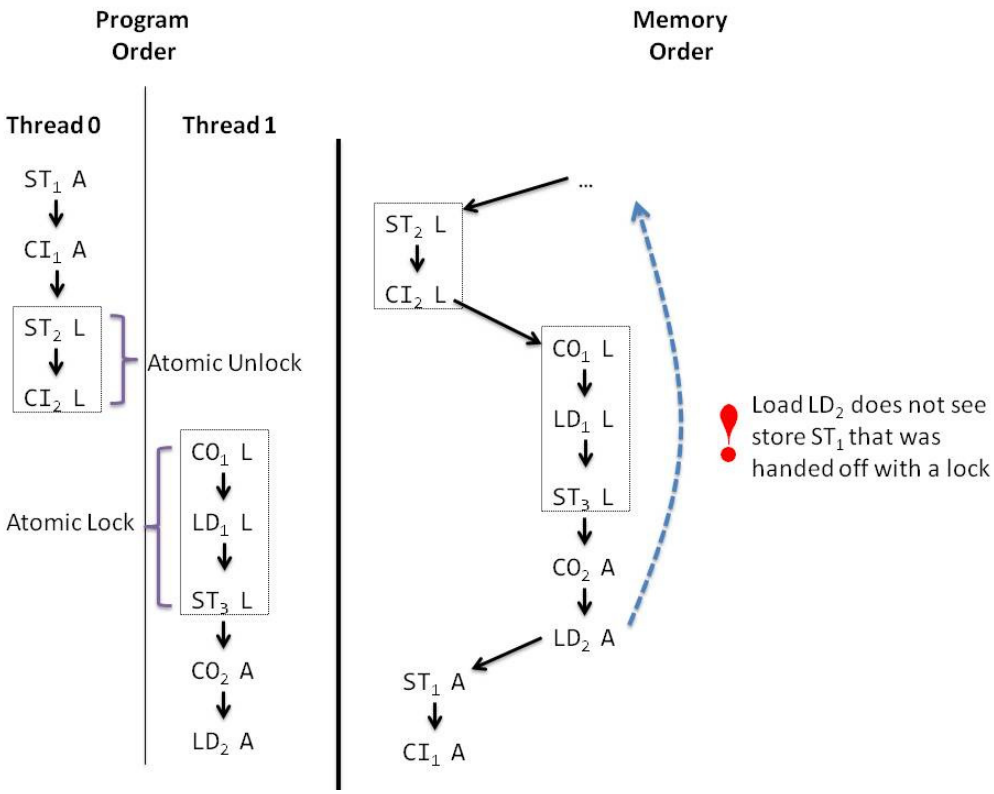

Figure A-5. Justification for common memory order constraint (3-15).

Assume A and L are in different segments, and the operations in dashed boxes are executed with a single atomic instruction. The constraint prevents the execution on the right.

The last common constraint, (3-15), is needed to ensure that threads can establish causality between two different segments (and by transitivity, among all segments). In particular, without this constraint thread would be unable to use synchronization variables located in a segment other than the one holding the data they are synchronizing. We show this in Figure A-5, which depicts two threads synchronizing on migratory data. The intended behavior is that after thread 0 passes the migratory object to thread 1 via a successful unlock/lock pair, the latest updates to the migratory object will be seen on thread 1. If the model allowed reordering between checkout and checkin, then CI_1 could appear after any other instruction in the example. In particular, if, as shown in the example, CI_1 is ordered in memory after LD_2, then the transitivity constraint (3-14) will not take effect, making it possible for ST_1 to occur after LD_2.

It is interesting to note that while the model requires a total order of checkin and checkout, implementations need not *directly* establish that order. This can be important because it allows an implementation to avoid a single serialization bottleneck when ordering checkout and checkin. More specifically, an implementation only needs to directly determine the order between checkout and checkin operations that establish a causal relationship between a store and load (via constraint (3-14)). From that partial order, a valid total order that still respect causality can be logically constructed by assigning arbitrary priority to concurrent operations.

Appendix B. Operational Definition of ASM Consistency

In this Appendix, we present an alternate formalization of ASM that is less mathematically rigorous but far more intuitive for programmers because it is described operationally. The model is called ASM-CVS because it describes the ASM in terms of the CVS revision control model that most programmers are already familiar with. In doing so, we give programmers an easy framework to reason about ASM when they may (intentionally) introduce data races in a program. ASM-CVS is loosely patterned after the framework used to describe x86-TSO [155].

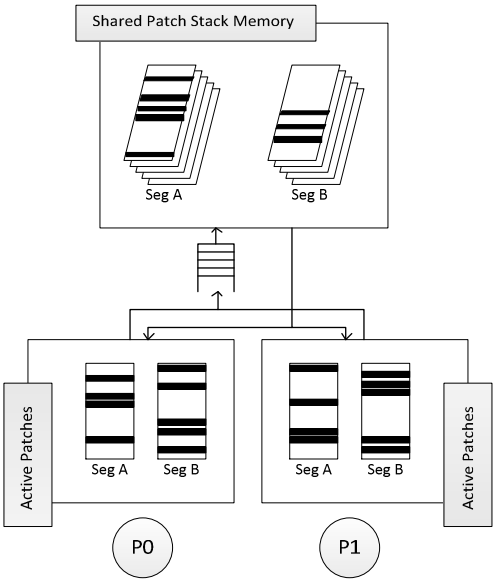

Figure B-1. ASM-CVS abstract machine model

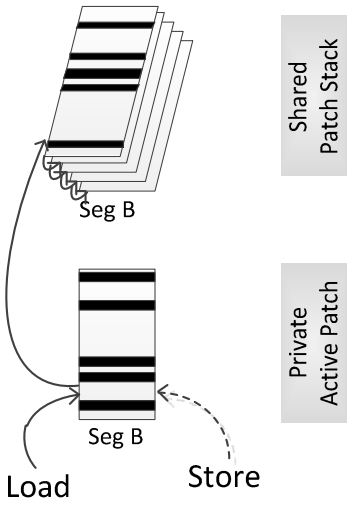

Figure B-2. Load search order and store path

ASM-CVS borrows concepts from revision control systems that most programmers are familiar with, most notably the concept of a patch. The ASM-CVS model adopts the abstract machine model shown in Figure B-1. The machine consists of a number of processors that each have a private *active patch* per segment and a single, shared *patch stack memory* that contains a stack of patches per segment. Each patch is a partial representation of a segment in a moment in

time. Locations within a particular patch may not have an assigned value, in which case a value from a patch lower in the stack applies, as shown in Figure B-2 (black bars indicate the location contains a value). The shared memory is accessed through a single, serialized queue shared by all processors and has five request types that are summarized in Table B-1. All shared memory requests complete atomically.

Table B-1. Request types for global patch memory

Request Type	Action
Find(L)	Locates the value of location L that is highest in the patch stack and returns it to the requesting processor
Write(L,V)	Writes the incoming value V to location L in the topmost patch on the stack
Create(S)	Creates a new, empty, patch on the top of the stack for segment S and returns a handle.
Finalize(P)	Finalizes the state of a patch P that was previously returned by a create request.
No-op	Does not change the state of shared memory, but does return a acknowledgement.

The behavior of the ASM-CVS system can be described with a set of four rules.

Load. A processor performs a load by first querying its local active patch. If no value is found, the processor issues a find request to shared memory and waits for a reply.

Store. A processor performs a store by updating the value of the store location in the current active patch.

Checkout. To complete a checkout, a processor clears all values from the current active patch, issues a no-op to shared memory, and then waits for an acknowledgement.

Checkin. A checkin first issues a create request to shared memory. Then (without waiting for an acknowledgement) continues to issue a write for each updated value in the current active patch. Finally, the processor issues a finalize request and then waits for an acknowledgement.

The checkin operation contains a subtlety that warrants further explanation. Note that a checkin operation is not atomic. It is possible for two checkins to happen simultaneously (i.e., there are two non-finalized patches on a stack at the same time). When this occurs, all processors performing the checkin will write their updates to the same patch that is on top of the stack. For all but one processor, this will not be the same patch that the processor created at the beginning of the checkin operation. If two updates conflict, the one written last will survive in the patch. Under the rules of DRF in Section 3.7, concurrent conflicting checkins cannot occur.

Appendix C. Absolute Results Data

Table C-1. Absolute runtime data in cycles.

	ASM-CMP	MOESI	MESI
barnes	366948221	362814975	362283277
fft	39603955	39381599	39543752
fmm	124341969	115340116	116908232
lu	60342424	60153644	59470330
mp3d	37112665	145666880	91364838
ocean	63864500	66281964	59473564
radix	11354012	10865271	11276008
water	29178278	28959303	28139132
cilksort	25439152	25599616	24348897
clu	76371699	70332671	67629875
heat2d	507510770	602723185	524958015
heat3d	562119165	568843202	561406260
matmul	319545693	311998712	289304806
uts_circ	666411554	66238228	663201037
uts_fixed	538880794	530310941	538226463

Table C-2. Absolute energy data for ASM-CMP in mJ

	L1 I	L1 D	L2	Link	Switch
barnes	142.7	187.3	236.2	344.9	258.1
fft	17.2	30.8	25.7	46.6	29.7
fmm	55.2	103.9	80.7	131.6	91.5
lu	22.7	29.7	39.4	70.8	45.4
mp3d	14.9	23.6	24.4	47.2	29.1
ocean	27.9	37.6	41.9	82.1	49.8
radix	4.7	7.7	7.5	17.3	9.3
water	14.6	20.2	18.9	29.3	21.4
cilksort	11.1	16.6	16.7	33.6	20.4
clu	31.1	36.5	49.6	75.6	56.4
heat2d	225.9	347.6	341.8	723.5	416.2
heat3d	243.7	371.9	364.8	640.4	417.7
matmul	125.5	159.6	208.8	370.5	243.4
uts_circ	320.2	528.9	436.3	813.5	511.4
uts_fixed	257.7	439.6	348.3	528.0	387.6

Table C-3. Absolute energy data for MOESI in mJ

	L1 I	L1 D	L2	Link	Switch
barnes	141.8	167.5	331.4	327.4	365.3
fft	17.3	27.7	36.6	35.5	43.1
fmm	52.0	90.2	105.3	104.1	116.1
lu	23.1	27.0	56.1	54.3	65.7
mp3d	56.2	62.6	141.2	131.5	150.4
ocean	29.4	35.1	62.2	59.8	74.2
radix	4.7	6.8	10.3	9.8	13.2
water	14.5	18.1	26.5	26.1	29.2
cilksort	11.5	15.3	24.1	23.1	29.1
clu	29.3	31.4	64.4	63.5	72.1
heat2d	225.2	309.3	478.9	458.0	583.3
heat3d	254.2	343.2	527.0	513.4	621.3
matmul	126.6	143.6	290.5	281.6	349.3
uts_circ	324.3	487.2	624.9	601.3	783.6
uts_fixed	225.1	393.5	484.9	478.6	538.6

Table C-4. Absolute energy data for MOESI in mJ

	L1 I	L1 D	L2	Link	Switch
barnes	141.4	167.0	330.3	327.0	304.9
fft	17.3	27.7	36.3	35.7	36.1
fmm	52.6	90.8	106.6	105.5	98.8
lu	22.8	26.4	54.8	53.7	53.9
mp3d	35.4	41.6	85.1	82.5	87.5
ocean	26.8	32.2	54.9	53.7	56.6
radix	4.8	6.9	10.5	10.2	11.4
water	14.6	20.2	18.9	29.3	21.4
cilksort	10.9	14.6	22.5	22.0	23.6
clu	28.1	30.1	61.7	61.0	58.9
heat2d	259.2	341.2	556.5	544.0	576.3
heat3d	246.8	334.7	514.7	506.7	507.8
matmul	117.0	133.4	266.0	261.1	261.1
uts_circ	319.4	482.2	610.9	598.5	595.0
uts_fixed	257.6	396.0	491.0	485.7	456.3