A Primer on Memory Consistency and Cache Coherence

Book *in* Synthesis Lectures on Computer Architecture · November 2011

DOI: 10.2200/S00346ED1V01Y201104CAC016 · Source: DBLP

3 authors:

Daniel Sorin
Duke University
170 PUBLICATIONS **6,771** CITATIONS
SEE PROFILE

Mark D. Hill
University of Wisconsin–Madison
259 PUBLICATIONS **24,266** CITATIONS
SEE PROFILE

David A. Wood
University of Wisconsin–Madison
253 PUBLICATIONS **19,081** CITATIONS
SEE PROFILE

Preliminary Draft

A Primer on Memory Consistency and Cache Coherence

Daniel J. Sorin, Mark D. Hill, and David A. Wood

Table of Contents

Preface

This primer is intended for readers who have encountered cache coherence and memory consistency informally, but now want to understand what they entail in more detail. This audience includes computing industry professionals as well as junior graduate students.

We expect our readers to be familiar with the basics of computer architecture. Remembering the details of Tomasulo's algorithm or similar details is unnecessary, but we do expect readers to understand issues like architectural state, dynamic instruction scheduling (out-of-order execution), and how caches are used to reduce average latencies to access storage structures.

The primary goal of this primer is to provide readers with a basic understanding of coherence and consistency. This understanding includes both the issues that must be solved as well as a variety of solutions. We present both high-level concepts as well as specific, concrete examples from real-world systems. A secondary goal of this primer is to make readers aware of just how complicated coherence and consistency are. If readers simply discover what it is that they do not know—without actually learning it—that discovery is still a substantial benefit. Furthermore, because these topics are so vast and so complicated, it is beyond the scope of this primer to cover them exhaustively. It is *not* a goal of this primer to cover all topics in depth, but rather to cover the basics and apprise the readers of what topics they may wish to pursue in more depth.

We owe many thanks for the help and support we have received during the development of this primer. We thank Blake Hechtman for implementing and testing (and debugging!) all of the coherence protocols in this primer. As the reader will soon discover, coherence protocols are complicated, and we would not have trusted any protocol that we had not tested, so Blake's work was tremendously valuable. Blake implemented and tested all of these protocols using the Wisconsin GEMS simulation infrastructure [http://www.cs.wisc.edu/gems/].

For reviewing early drafts of this primer and for helpful discussions regarding various topics within the primer, we gratefully thank Trey Cain and Milo Martin. For providing additional feedback on the primer, we thank Newsha Ardalani, Arkaprava Basu, Brad Beckmann, Bob Cypher, Joe Devietti, Sandip Govind Dhoot, Alex Edelsburg, Jayneel Gandhi, Dan Gibson, Marisabel Guevara, Gagan Gupta, Blake Hechtman, Derek Hower, Zachary Marzec, Hiran Mayukh, Ralph Nathan, Marc Orr, Vijay Sathish, Abhirami Senthilkumaran, Simha Sethumadhavan, Venkatanathan Varadarajan, Derek Williams, and Meng

Zhang. While our reviewers provided great feedback, they may or may not agree with all of the final contents of this primer.

This work is supported in part by the National Science Foundation (CNS-0551401, CNS-0720565, CCF-0916725, CCF-0444516, and CCF-0811290), Sandia/DOE (#MSN123960/DOE890426), Semiconductor Research Corporation (contract 2009-HJ-1881), and the University of Wisconsin (Kellett Award to Hill). The views expressed herein are not necessarily those of the NSF, Sandia, DOE, or SRC.

Dan thanks Deborah, Jason, and Julie for their love and for putting up with him taking the time to work on another synthesis lecture. Dan thanks his Uncle Sol for helping inspire him to be an engineer in the first place. Lastly, Dan dedicates this book to the memory of Rusty Sneiderman, a treasured friend of thirty years who will be dearly missed by everyone who was lucky enough to have known him.

Mark wishes to thank Sue, Nicole, and Gregory for their love and support.

David thanks his coauthors for putting up with his deadline-challenged work style, his parents Roger and Ann Wood for inspiring him to be a second-generation Computer Sciences professor, and Jane, Alex, and Zach for helping me remember what life is all about.

Chapter 1

Introduction to Consistency and Coherence

Many modern computer systems and most multicore chips (chip multiprocessors) support shared memory in hardware. In a shared memory system, each of the processor cores may read and write to a single shared address space. These designs seek various goodness properties, such as high performance, low power, and low cost. Of course, it is not valuable to provide these goodness properties without first providing correctness. Correct shared memory seems intuitive at a hand-wave level, but, as this lecture will help show, there are subtle issues in even defining what it means for a shared memory system to be correct, as well as many subtle corner cases in designing a correct shared memory implementation. Moreover, these subtleties must be mastered in hardware implementations where bug fixes are expensive. Even academics should master these subtleties to make it more likely that their proposed designs will work.

We and many others find it useful to separate shared memory correctness into two sub-issues: *consistency and coherence*. Computer systems are not required to make this separation, but we find it helps to divide and conquer complex problems, and this separation prevails in many real shared memory implementations.

It is the job of consistency (memory consistency, memory consistency model, or memory model) to define shared memory correctness. Consistency definitions provide rules about loads and stores (or memory reads and writes) and how they act upon memory. Ideally, consistency definitions would be simple and easy to understand. However, defining what it means for shared memory to behave correctly is more subtle than defining the correct behavior of, for example, a single-threaded processor core. The correctness criterion for a single processor core partitions behavior between one correct result and many incorrect alternatives. This is because the processor's architecture mandates that the execution of a thread transforms a given input state into a single well-defined output state, even on an out-of-order core. Shared memory consistency models, however, concern the loads and stores of multiple threads and usually allow many correct executions while disallowing many (more) incorrect ones. The possibility of multiple correct executions is due to the ISA allowing multiple threads to execute concurrently, often with many possible legal interleavings of instructions from different threads. The multitude of correct executions complicates the erstwhile simple challenge of determining whether an execution is correct. Nevertheless, consistency must be mastered to implement shared memory, and, in some cases, to write correct programs that use it.

Unlike consistency, coherence (or cache coherence) is neither visible to software nor required. However, as part of supporting a consistency model, the vast majority of shared memory systems implement a coherence protocol that provides coherence. Coherence seeks to make the caches of a shared-memory system as functionally invisible as the caches in a single-core system. Correct coherence ensures that a programmer cannot determine whether and where a system has caches by analyzing the results of loads and stores. This is because correct coherence ensures that the caches never enable new or different *functional* behavior. (Programmers may still be able to infer likely cache structure using *timing* information.)

In most systems, coherence protocols play an important role in providing consistency. Thus, even though consistency is the first major topic of this primer, we begin in Chapter 2 with a brief introduction to coherence. The goal of this chapter is to explain enough about coherence to understand how consistency models interact with coherent caches, but not to explore specific coherence protocols or implementations, which are topics we defer until the second portion of this primer in Chapter 6-Chapter 9. In Chapter 2, we define coherence using the single-writer-multiple-reader (SWMR) invariant. SWMR requires that, at any given time, a memory location is either cached for writing (and reading) at one cache or cached only for reading at zero to many caches.

1.1 Consistency (a.k.a., Memory Consistency, Memory Consistency Model, or Memory Model)

Consistency models define correct shared memory behavior in terms of loads and stores (memory reads and writes), without reference to caches or coherence. To gain some real-world intuition on why we need consistency models, consider a university that posts its course schedule online. Assume that the Computer Architecture course is originally scheduled to be in Room 152. The day before classes begin, the university registrar decides to move the class to Room 252. The registrar sends an email message asking the web site administrator to update the online schedule, and a few minutes later the registrar sends a text message to all registered students to check the newly updated schedule. It is not hard to imagine a scenario—if, say, the web site administrator is too busy to post the update immediately—in which a diligent student receives the text message, immediately checks the online schedule, and still observes the (old) class location Room 152. Even though the online schedule is eventually updated to Room 252 and the registrar performed the “writes” in the correct order, this diligent student observed them in a different order and thus went to the wrong room. A consistency model defines whether this behavior is correct (and thus whether a user must take other action to achieve the desired outcome) or incorrect (in which case the system must preclude these reorderings).

Although this contrived example used multiple media, similar behavior can happen in shared memory hardware with out-of-order processor cores, write buffers, prefetching, and multiple cache banks. Thus, we need to define shared memory correctness—that is, which shared memory behaviors are allowed—so that programmers know what to expect and implementors know the limits to what they can provide.

Shared memory correctness is specified by a memory consistency model or, more simply, a memory model. The memory model specifies the allowed behavior of multithreaded programs executing with shared memory. For a multithreaded program executing with specific input data, the memory model specifies what values dynamic loads may return and what are the possible final states of memory. Unlike single-threaded execution, multiple correct behaviors are usually allowed, making understanding memory consistency models subtle.

Chapter 3 introduces the concept of memory consistency models and presents sequential consistency (SC), the strongest and most intuitive consistency model. The chapter begins by motivating the need to specify shared memory behavior and precisely defines what a memory consistency model is. It next delves into the intuitive SC model, which states that a multithreaded execution should look like an interleaving of the sequential executions of each constituent thread, as if the threads were time-multiplexed on a single-core processor. Beyond this intuition, the chapter formalizes SC and explores implementing SC with coherence in both simple and aggressive ways, culminating with a MIPS R10000 case study.

In Chapter 4, we move beyond SC and focus on the memory consistency model implemented by x86 and SPARC systems. This consistency model, called total store order (TSO), is motivated by the desire to use first-in-first-out write buffers to hold the results of committed stores before writing the results to the caches. This optimization violates SC, yet promises enough performance benefit to inspire architectures to define TSO, which permits this optimization. In this chapter, we show how to formalize TSO from our SC formalization, how TSO affects implementations, and how SC and TSO compare.

Finally, Chapter 5 introduces “relaxed” or “weak” memory consistency models. It motivates these models by showing that most memory orderings in strong models are unnecessary. If a thread updates ten data items and then a synchronization flag, programmers usually do not care if the data items are updated in order with respect to each other but only that all data items are updated before the flag is updated. Relaxed models seek to capture this increased ordering flexibility to get higher performance or a simpler implementation. After providing this motivation, the chapter develops an example relaxed consistency model, called XC, wherein programmers get order only when they ask for it with a FENCE instruction (e.g., a FENCE after the last data update but before the flag write). The chapter then extends the formalism of the previous two chapters to handle XC and discusses how to implement XC (with considerable reordering between the cores and the coherence protocol). The chapter then discusses a way in which many programmers can avoid thinking about relaxed models directly: If they add enough FENCEs to ensure their program is data-race free (DRF), then most relaxed models will appear SC. With "SC for DRF," programmers can get both the (relatively) simple correctness model of SC with the (relative) higher performance of XC. For those who want to reason more deeply, the chapter concludes by distinguishing acquires from releases, discussing write atomicity and causality, pointing to commercial examples (including an IBM Power case study), and touching upon high-level language models (Java and C++).

Returning to the real-world consistency example of the class schedule, we can observe that the combination of an email system, a human web administrator, and a text-messaging system represents an extremely weak consistency model. To prevent the problem of a diligent student going to the wrong room, the university registrar needed to perform a FENCE operation after her email, to ensure that the online schedule was updated before sending the text message.

1.2 Coherence (a.k.a., Cache Coherence)

Unless care is taken, a coherence problem can arise if multiple actors (e.g., multiple cores) have access to multiple copies of a datum (e.g., in multiple caches) and at least one access is a write. Consider an example that is similar to the memory consistency example. A student checks the online schedule of courses and observes that the Computer Architecture course is being held in Room 152 (reads the datum) and copies this information into her notebook (caches the datum). Subsequently, the university registrar decides to move the class to Room 252 and updates the online schedule (writes to the datum). The student's copy of the datum is now stale, and we have an incoherent situation. If she goes to Room 152, she will fail to find her class. Examples of incoherence from the world of computing, but not including computer architecture, include stale Web caches and programmers using un-updated code repositories.

Access to stale data (incoherence) is prevented using a coherence protocol, which is a set of rules implemented by the distributed set of actors within a system. Coherence protocols come in many variants, but follow a few themes, as developed in Chapter 6-Chapter 9.

Chapter 6 presents the big picture of cache coherence protocols and sets the stage for the subsequent chapters on specific coherence protocols. This chapter covers issues shared by most coherence protocols, including the distributed operations of cache controllers and memory controllers and the common MOESI coherence states: modified (M), owned (O), exclusive (E), shared (S), and invalid (I). Importantly, this chapter also presents our table-driven methodology for presenting protocols with both stable (e.g., MOESI) and transient coherence states. Transient states are required in real implementations, because modern systems rarely permit atomic transitions from one stable state to another (e.g., a read miss in state Invalid will spend some time waiting for a data response before it can enter state Shared). Much of the real complexity in coherence protocols hides in the transient states, similar to how much of processor core complexity hides in micro-architectural states.

Chapter 7 covers snooping cache coherence protocols, which dominated the commercial market until fairly recently. At the hand-wave level, snooping protocols are simple. When a cache miss occurs, a core's cache controller arbitrates for a shared bus and broadcasts its request. The shared bus ensures that all controllers observe all requests in the same order and thus all controllers can coordinate their individual, distributed actions to ensure that they maintain a globally consistent state. Snooping gets complicated, however, because systems may use multiple buses and modern buses do not atomically handle requests. Modern buses have queues for arbitration and can send responses that are unicast, delayed by pipelining, or out-of-order. All of these features lead to more transient coherence states. Chapter 7 concludes with case studies of the Sun UltraEnterprise E10000 and the IBM Power5.

Chapter 8 delves into directory cache coherence protocols that offer the promise of scaling to more processor cores and other actors than snooping protocols that rely on broadcast. There is a joke that all problems in computer science can be solved with a level of indirection. Directory protocols support this joke: A cache miss requests a memory location from the next level cache (or memory) controller, which maintains a directory that tracks which caches hold which locations. Based on the directory entry for the requested memory location, the controller sends a response message to the requestor or forwards the request message to one or more actors currently caching the memory location. Each message typically has one destination (i.e., no broadcast or multicast), but transient coherence states abound as transitions from one stable coherence state to another stable one can generate a number of messages proportional to the number of actors in the system. This chapter starts with a basic directory protocol and then refines it to handle the MOESI states E and O, distributed directories, less stalling of requests, approximate directory entry representations, and more. The chapter also explores the design of the directory itself, including directory caching techniques. The chapter concludes with case studies of the old SGI Origin 2000 and the newer AMD HyperTransport, HyperTransport Assist, and Intel QuickPath Interconnect (QPI).

Chapter 9 deals with some, but not all, of the advanced topics in coherence. For ease of explanation, the prior chapters on coherence intentionally restrict themselves to the simplest system models needed to explain the fundamental issues. Chapter 9 delves into more complicated system models and optimizations, with a focus on issues that are common to both snooping and directory protocols. Initial topics include dealing with instruction caches, multi-level caches, write-through caches, translation lookaside buffers (TLBs), coherent direct memory access (DMA), virtual caches, and hierarchical coherence protocols. Finally, the chapter delves into performance optimizations (e.g., targeting migratory sharing and false sharing) and directly maintaining the SWMR invariant with token coherence.

1.3 A Consistency and Coherence Quiz

It can be easy to convince oneself that one's knowledge of consistency and coherence is sufficient and that reading this primer is not necessary. To test whether this is the case we offer this pop quiz.

Question 1: In a system that maintains sequential consistency, a core must issue coherence requests in program order. True or false? (Answer is in Section 3.8)

Question 2: The memory consistency model specifies the legal orderings of coherence transactions. True or false? (Section 3.8)

Question 3: To perform an atomic read-modify-write instruction (e.g., test-and-set), a core must always communicate with the other cores. True or false? (Section 3.9)

Question 4: In a TSO system with multithreaded cores, threads may bypass values out of the write buffer, regardless of which thread wrote the value. True or false? (Section 4.4)

Question 5: A programmer who writes properly synchronized code relative to the high-level language's consistency model (e.g,. Java) does not need to consider the architecture's memory consistency model. True or false? (Section 5.9)

Question 6: In an MSI snooping protocol, a cache block may only be in one of three coherence states. True or false? (Section 7.2)

Question 7: A snooping cache coherence protocol requires the cores to communicate on a bus. True or false? (Section 7.6)

Even though the answers are provided later in this primer, we encourage readers to try to answer the questions before looking ahead at the answers.

1.4 What this Primer Does NOT Do

This lecture is intended to be a primer on coherence and consistency. We expect this material could be covered in a graduate class in about nine 75-minute classes, e.g., one lecture per Chapter 2 to Chapter 9 plus one lecture for advanced material).

For this purpose, there are many things the primer does *not* cover. Some of these include:

- Synchronization. Coherence makes caches invisible. Consistency can make shared memory look like a single memory module. Nevertheless, programmers will probably need locks, barriers, and other synchronization techniques to make their programs useful.
- Commercial Relaxed Consistency Models. This primer does not cover all the subtleties of the ARM and PowerPC memory models, but does describe which mechanisms they provide to enforce order.
- Parallel programming. This primer does not discuss parallel programming models, methodologies, or tools.