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.

Designing and evaluating a correct shared memory system requires an architect to understand *memory consistency* and *cache coherence*, the two topics of this primer. Memory consistency (consistency, memory consistency model, or memory model) is a precise, architecturally-visible definition of 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.

The microarchitecture—the hardware design of the processor cores and the shared memory system—must enforce the desired consistency model. As part of this consistency model support, the hardware provides cache coherence (or coherence). In a shared-memory system with caches, the cached values can potentially become out-of-date (or incoherent) when one of the processors updates its cached value. Coherence seeks to make the caches of a shared-memory system as functionally invisible as the caches in a single-core system; it does so by propagating a processor's write to other processors' caches. It is worth stressing that unlike consistency which is an architectural specification that defines shared memory correctness, coherence is a means to supporting a consistency model.

Even though consistency is the first major topic of this primer, we begin in Chapter 2 with a brief introduction to coherence because coherence protocols play an important role in providing consistency. 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 Chapters 6–9.

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 e-mail 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, optionally, what possible final states of the memory are. 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 historical 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 (relatively) 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, observes that the Computer Architecture course is being held in Room 152 (reads the datum), and copies this information into her calendar app in her mobile phone (caches the datum). Subsequently, the university registrar decides to move the class to Room 252, updates the online schedule (writes to the datum) and informs the students via a text message. 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 Chapters 6–9. Essentially, all of the variants make one processor's write visible to the other processors by propagating the write to all caches, i.e., keeping the calendar in

sync with the online schedule. But protocols differ in *when* and *how* the syncing happens. There are two major classes of coherence protocols. In the first approach, the coherence protocol ensures that writes are propagated to the caches synchronously. When the online schedule in updated, the coherence protocol ensures that the student's calendar is updated as well. In the second approach, the coherence protocol propagates writes to the caches asynchronously, while still honoring the consistency model. The coherence protocol does not guarantee that when the online schedule is updated, the new value will have propagated to the student's calendar as well; however, the protocol does ensure that the new value is propagated before the text message reaches her mobile phone. This primer focuses on the first class of the coherence protocols (Chapters 6–9) while Chapter 10 discusses the emerging second class.

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 initially dominated the commercial market. 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, multilevel 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 CONSISTENCY AND COHERENCE FOR HETEROGENEOUS SYSTEMS

Modern computer systems are predominantly heterogeneous. A mobile phone processor today not only contains a multicore CPU, it also has a GPU and other accelerators (e.g., neural network hardware). In the quest for programmability, such heterogeneous systems are starting to support shared memory. Chapter 10 deals with consistency and coherence for such heterogeneous processors.

The chapter starts by focusing on GPUs, arguably the most popular among accelerators today. The chapter observes that GPUs originally chose not to support hardware cache coherence, since GPUs are designed for embarrassingly parallel graphics workloads that do not synchronize or share data all that much. However, the absence of hardware cache coherence leads to programmability and/or performance challenges when GPUs are used for general-purpose workloads with fine-grained synchronization and data sharing. The chapter discusses in detail some of

the promising coherence alternatives that overcome these limitations—in particular, explaining why the candidate protocols enforce the consistency model directly rather than implementing coherence in a consistency-agnostic manner. The chapter concludes with a brief discussion on consistency and coherence across CPUs and the accelerators.

1.4 SPECIFYING AND VALIDATING MEMORY CONSISTENCY MODELS AND CACHE COHERENCE

Consistency models and coherence protocols are complex and subtle. Yet, this complexity must be managed to ensure that multicores are programmable. To achieve this goal, it is critical that consistency models are specified formally. A formal specification would enable programmers to clearly and exhaustively (with tool support) understand what behaviors are permitted by the memory model and what behaviors are not. Second, a precise formal specification is mandatory for validating implementations.

Chapter 11 starts by discussing two methods for specifying systems—axiomatic and operational—focusing on how these methods can be applied for consistency models and coherence protocols. Then the chapter goes over techniques for validating implementations—including processor pipeline and coherence protocol implementations—against their specification. The chapter focuses on both formal methods and informal testing.

1.5 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)

Question 8: GPUs do not support hardware cache coherence. Therefore, they are unable to enforce a memory consistency model. True or False? (Section 10.1).

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.

1.6 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. Readers are referred to the Synthesis Lecture on Shared-memory synchronization [2].

- 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.

- Consistency in distributed systems. This primer restricts itself to consistency within a shared memory multicore, and does not cover consistency models and their enforcement for a general distributed system. Readers are referred to the Synthesis lectures on Database Replication [1] and Quorum Systems [3].

REFERENCES

[1] B. Kemme, R. Jiménez-Peris, and M. Patiño-Martínez. *Database Replication*. Synthesis Lectures on Data Management. Morgan & Claypool Publishers, 2010. 8

[2] M. L. Scott. *Shared-Memory Synchronization*. Synthesis Lectures on Computer Architecture. Morgan & Claypool Publishers, 2013. 8

[3] M. Vukolic. *Quorum Systems: With Applications to Storage and Consensus*. Synthesis Lectures on Distributed Computing Theory. Morgan & Claypool Publishers, 2012. 8

CHAPTER 2

Coherence Basics

In this chapter, we introduce enough about cache coherence to understand how consistency models interact with caches. We start in Section 2.1 by presenting the system model that we consider throughout this primer. To simplify the exposition in this chapter and the following chapters, we select the simplest possible system model that is sufficient for illustrating the important issues; we defer until Chapter 9 issues related to more complicated system models. Section 2.2 explains the cache coherence problem that must be solved and how the possibility of incoherence arises. Section 2.3 precisely defines cache coherence.

2.1 BASELINE SYSTEM MODEL

In this primer, we consider systems with multiple processor cores that share memory. That is, all cores can perform loads and stores to all (physical) addresses. The baseline system model includes a single multicore processor chip and off-chip main memory, as illustrated in Figure 2.1. The multicore processor chip consists of multiple single-threaded cores, each of which has its own private data cache, and a last-level cache (LLC) that is shared by all cores. Throughout this primer, when we use the term "cache," we are referring to a core's private data cache and not the LLC. Each core's data cache is accessed with physical addresses and is write-back. The cores and the LLC communicate with each other over an interconnection network. The LLC, despite being on the processor chip, is logically a "memory-side cache" and thus does not introduce another level of coherence issues. The LLC is logically just in front of the memory and serves to reduce the average latency of memory accesses and increase the memory's effective bandwidth. The LLC also serves as an on-chip memory controller.

This baseline system model omits many features that are common but that are not required for purposes of most of this primer. These features include instruction caches, multiple-level caches, caches shared among multiple cores, virtually addressed caches, TLBs, and coherent direct memory access (DMA). The baseline system model also omits the possibility of multiple multicore chips. We will discuss all of these features later, but for now, they would add unnecessary complexity.

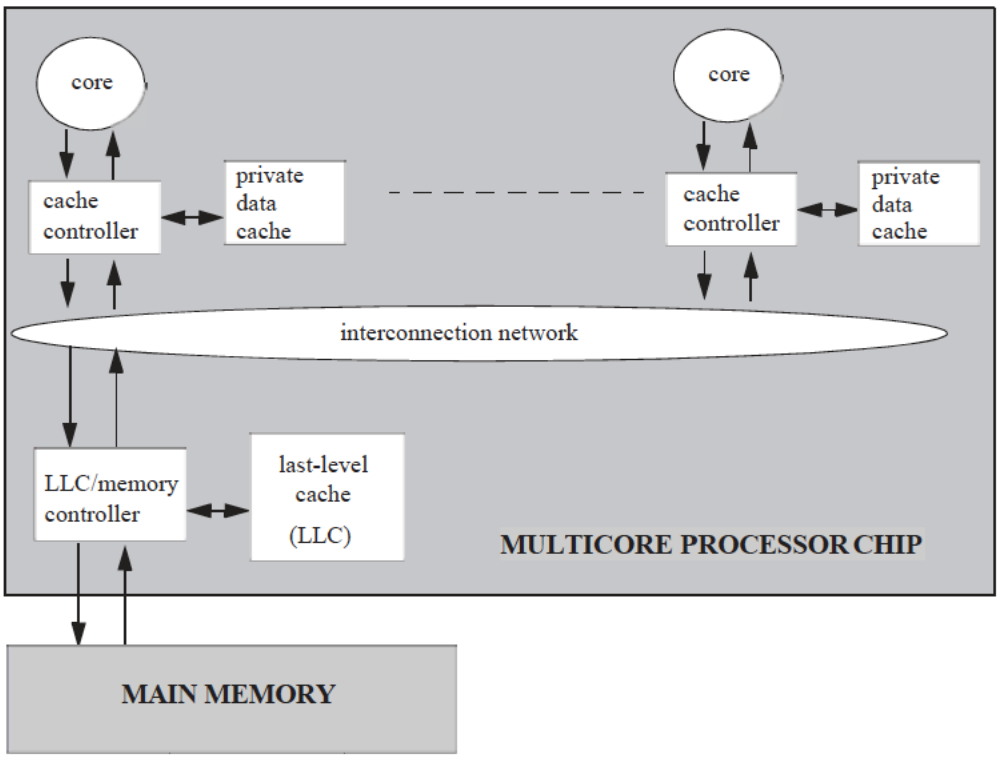

Figure 2.1: Baseline system model used throughout this primer.

2.2 THE PROBLEM: HOW INCOHERENCE COULD POSSIBLY OCCUR

The possibility of incoherence arises only because of one fundamental issue: there exist multiple actors with access to caches and memory. In modern systems, these actors are processor cores, DMA engines, and external devices that can read and/or write to caches and memory. In the rest of this primer, we generally focus on actors that are cores, but it is worth keeping in mind that other actors may exist.

Table 2.1 illustrates a simple example of incoherence. Initially, memory location A has the value 42 in memory as well as each of the core's local caches. At time 1, Core 1 changes the value at memory location A from 42 to 43 in its cache, making Core 2's value of A in its cache stale. Core 2 executes a while loop loading, repeatedly, the (stale) value of A from its local cache. Clearly, this is an example of incoherence as the store from Core 1 has not not been made visible to Core 2 and consequently C2 is stuck in the while loop.

Table 2.1: Example of incoherence. Assume the value of memory at memory location A is initially 42 and cached in the local caches of both cores.

Time	Core C1	Core C2
1	S1: A = 43;	L1: while (A == 42);
2		L2: while (A == 42);
3		L3: while (A == 42);
4		...
n		Ln: while (A == 42);

To prevent incoherence, the system must implement a *cache coherence protocol* that makes the store from Core 1 visible to Core 2. The design and implementation of these cache coherence protocols are the main topics of Chapter 7 through Chapter 9.

2.3 THE CACHE COHERENCE INTERFACE

Informally, a coherence protocol must ensure that writes are made visible to all processors. In this section, we will more formally understand coherence protocols through the abstract interfaces they expose.

The processor cores interact with the coherence protocol through a coherence interface (Figure 2.2) that provides two methods: (1) a *read-request* method that takes in a memory location as the parameter and returns a value; (2) a *write-request* method takes in a memory location and a value (to be written) as parameters and returns an acknowledgment.

There are a number of coherence protocols that have appeared in the literature and been employed in real processors. We classify these protocols into two categories based on the nature of their coherence interfaces—specifically, based on whether there is a clean separation of coherence from the consistency model or whether they are indivisible.

Consistency-agnostic coherence. In the first category, a write is made visible to all other cores before returning. Because writes are propagated synchronously, the first category presents an interface that is identical to that of an atomic memory system (with no caches). Thus, any sub-system that interacts with the coherence protocol—e.g., the processor core pipeline—can assume it is interacting with an atomic memory system with no caches present. From a consistency enforcement perspective, this coherence interface enables a nice separation of concerns. The cache coherence protocol abstracts away the caches completely and presents an illusion of atomic memory—it is as if the caches are removed and only the memory is contained within the coherence box (Figure 2.2)—while the processor core pipeline enforces the orderings mandated by the consistency model specification.

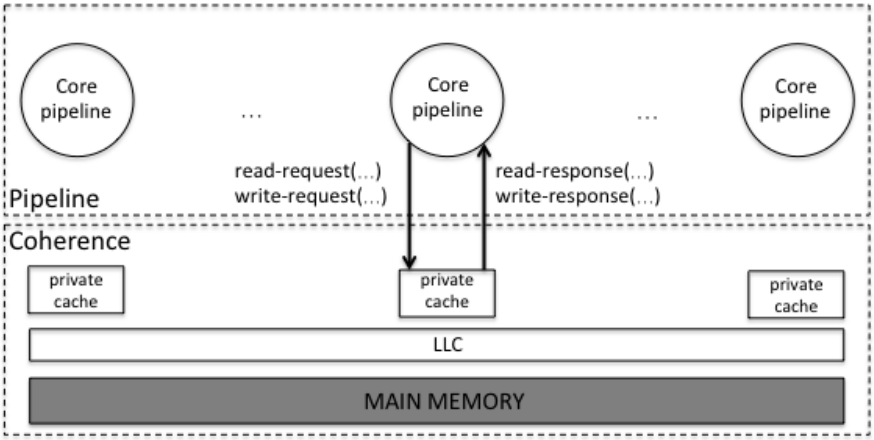

Figure 2.2: The pipeline-coherence interface.

Consistency-directed coherence. In the second more-recent category, writes are propagated asynchronously—a write can thus return before it has been made visible to all processors, thus allowing for stale values (in real time) to be observed. However, in order to correctly enforce consistency, coherence protocols in this class must ensure that the order in which writes are eventually made visible adheres to the ordering rules mandated by the consistency model. Referring back to Figure 2.2, both the pipeline and the coherence protocol enforce the orderings mandated by the consistency model. This second category emerged to support throughput-based general-purpose graphics processing units (GP-GPUs) and gained prominence after the publication of the first edition of this primer.

The primer (and the rest of the chapter) focuses on the first class of coherence protocols. We discuss the second class of coherence protocols in the context of heterogeneous coherence (chapter 10).

2.4 (CONSISTENCY-AGNOSTIC) COHERENCE INVARIANTS

What invariants must a coherence protocol satisfy to make the caches invisible and present an abstraction of an atomic memory system?

There are several definitions of coherence that have appeared in textbooks and in published papers, and we do not wish to present all of them. Instead, we present the definition we prefer for its insight into the design of coherence protocols. In the sidebar, we discuss alternative definitions and how they relate to our preferred definition.

We define coherence through the *single-writer–multiple-reader (SWMR)* invariant. For any given memory location, at any given moment in time, there is either a single core that may write it (and that may also read it) or some number of cores that may read it. Thus, there is never a time when a given memory location may be written by one core and simultaneously either read or written by any other cores. Another way to view this definition is to consider, for each memory location, that the memory location's lifetime is divided up into epochs. In each epoch, either a single core has read-write access or some number of cores (possibly zero) have read-only access. Figure 2.3 illustrates the lifetime of an example memory location, divided into four epochs that maintain the SWMR invariant.

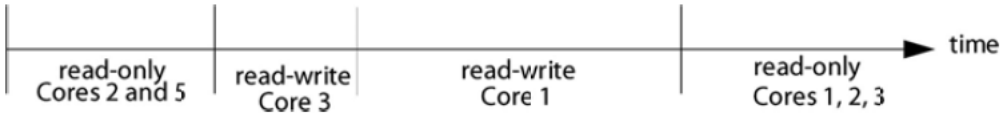

Figure 2.3: Dividing a given memory location's lifetime into epochs.

In addition to the SWMR invariant, coherence requires that the value of a given memory location is propagated correctly. To explain why values matter, let us reconsider the example in Figure 2.3. Even though the SWMR invariant holds, if during the first read-only epoch Cores 2 and 5 can read different values, then the system is not coherent. Similarly, the system is incoherent if Core 1 fails to read the last value written by Core 3 during its read-write epoch or any of Cores 1, 2, or 3 fail to read the last write performed by Core 1 during its read-write epoch.

Thus, the definition of coherence must augment the SWMR invariant with a data value invariant that pertains to how values are propagated from one epoch to the next. This invariant states that the value of a memory location at the start of an epoch is the same as the value of the memory location at the end of its last read-write epoch.

There are other interpretations of these invariants that are equivalent. One notable example [5] interpreted the SMWR invariants in terms of tokens. The invariants are as follows. For each memory location, there exists a fixed number of tokens that is at least as large as the number of cores. If a core has all of the tokens, it may write the memory location. If a core has one or more tokens, it may read the memory location. At any given time, it is thus impossible for one core to be writing the memory location while any other core is reading or writing it.

Coherence invariants

1. **Single-Writer, Multiple-Read (SWMR) Invariant.** For any memory location A, at any given (logical) time, there exists only a single core that may write to A (and can also read it) or some number of cores that may only read A.
2. **Data-Value Invariant.** The value of the memory location at the start of an epoch is the same as the value of the memory location at the end of the its last read-write epoch.

2.4.1 MAINTAINING THE COHERENCE INVARIANTS

The coherence invariants presented in the previous section provide some intuition into how coherence protocols work. The vast majority of coherence protocols, called "invalidate protocols," are designed explicitly to maintain these invariants. If a core wants to read a memory location, it sends messages to the other cores to obtain the current value of the memory location and to ensure that no other cores have cached copies of the memory location in a read-write state. These messages end any active read-write epoch and begin a read-only epoch. If a core wants to write to a memory location, it sends messages to the other cores to obtain the current value of the memory location, if it does not already have a valid read-only cached copy, and to ensure that no other cores have cached copies of the memory location in either read-only or read-write states. These messages end any active read-write or read-only epoch and begin a new read-write epoch. This primer's chapters on cache coherence (Chapters 6–9) expand greatly upon this abstract description of invalidate protocols, but the basic intuition remains the same.

2.4.2 THE GRANULARITY OF COHERENCE

A core can perform loads and stores at various granularities, often ranging from 1 to 64 bytes. In theory, coherence could be performed at the finest load/store granularity. However, in practice, coherence is usually maintained at the granularity of cache blocks. That is, the hardware enforces coherence on a cache block by cache block basis. In practice, the SWMR invariant is likely to be that, for any *block* of memory, there is either a single writer or some number of readers. In typical systems, it is not possible for one core to be writing to the first byte of a block while another core is writing to another byte within that block. Although cache-block granularity is common, and it is what we assume throughout the rest of this primer, one should be aware that there have been protocols that have maintained coherence at finer and coarser granularities.

Sidebar: Consistency-Like Definitions of Coherence

Our preferred definition of coherence defines it from an implementation perspective—specifying hardware-enforced invariants regarding the access permissions of different cores to a memory location and the data values passed between cores.

There exists another class of definitions that defines coherence from a programmer's perspective, similar to how memory consistency models specify architecturally visible orderings of loads and stores.

One consistency-like approach to specifying coherence is related to the definition of sequential consistency. Sequential consistency (SC), a memory consistency model that we discuss in great depth in Chapter 3, specifies that the system must appear to execute all threads' loads and stores to all memory locations in a total order that respects the program order of each thread. Each load gets the value of the most recent store in that total order. A definition of coherence that is analogous to the definition of SC is that a coherent system must appear to execute all threads' loads and stores to a single memory location in a total order that respects the program order of each thread. This definition highlights an important distinction between coherence and consistency in the literature: coherence is specified on a per-memory location basis, whereas consistency is specified with respect to all memory locations. It is worth noting that any coherence protocol that satisfies the SWMR and data-value invariants (combined with a pipeline that does not reorder accesses to any specific location) is also guaranteed to satisfy this consistency-like definition of coherence. (However, the converse is not necessarily true).

Another definition [1, 2] of coherence defines coherence with two invariants: (1) every store is eventually made visible to all cores and (2) writes to the same memory location are serialized (i.e., observed in the same order by all cores). IBM takes a similar view in the Power architecture [4], in part to facilitate implementations in which a sequence of stores by one core may have reached some cores (their values visible to loads by those cores) but not other cores. Invariant 2 is equivalent to the consistency-like definition we described earlier. In contrast to invariant 2, which is a *safety* invariant (bad things must not happen), invariant 1 is a *liveness* invariant (good things must eventually happen).

Another definition of coherence, as specified by Hennessy and Patterson [3], consists of three invariants: (1) a load to memory location A by a core obtains the value of the previous store to A by that core, unless another core has stored to A in between, (2) a load to A obtains the value of a store S to A by another core if S and the load "are sufficiently separated in time" and if no

other store occurred between S and the load, and (3) stores to the same memory location are serialized (same as invariant #2 in the previous definition). Like the previous definition, these set of invariants capture both safety and liveness.

2.4.3 WHEN IS COHERENCE RELEVANT?

The definition of coherence—regardless of which definition we choose—is relevant only in certain situations, and architects must be aware of when it pertains and when it does not. We now discuss two important issues:

- Coherence pertains to *all* storage structures that hold blocks from the shared address space. These structures include the L1 data cache, L2 cache, shared last-level cache (LLC), and main memory. These structures also include the L1 instruction cache and translation lookaside buffers (TLBs).[1]
- Coherence is not *directly* visible to the programmer. Rather, the processor pipeline and coherence protocol jointly enforce the consistency model—and it is only the consistency model that is visible to the programmer.

REFERENCES

[1] K. Gharachorloo. Memory consistency models for shared-memory multiprocessors. Ph.D. thesis, Computer System Laboratory, Stanford University, December 1995. 17

[2] K. Gharachorloo, D. Lenoski, J. Laudon, P. Gibbons, A. Gupta, and J. Hennessy. Memory consistency and event ordering in scalable shared-memory. In *Proc. of the 17th Annual International Symposium on Computer Architecture*, pp. 15–26, May 1990. 17

[3] J. L. Hennessy and D. A. Patterson. *Computer Architecture: A Quantitative Approach*, 4th ed. Morgan Kaufmann, 2007. 17

[4] IBM. Power ISA Version 2.06 Revision B. `http://www.power.org/resources/downloads/PowerISA_V2.06B_V2_PUBLIC.pdf`, July 2010. 17

[5] M. M. K. Martin, M. D. Hill, and D. A. Wood. Token coherence: Decoupling performance and correctness. In *Proc. of the 30th Annual International Symposium on Computer Architecture*, June 2003. DOI: 10.1109/ISCA.2003.1206999 15

[1] In some architectures, the TLB can hold mappings that are not strictly copies of blocks in shared memory.

CHAPTER 3

Memory Consistency Motivation and Sequential Consistency

This chapter delves into memory consistency models (a.k.a. memory models) that define the behavior of shared memory systems for programmers and implementors. These models define correctness so that programmers know what to expect and implementors know what to provide. We first motivate the need to define memory behavior (Section 3.1), say what a memory consistency model should do (Section 3.2), and compare and contrast consistency and coherence (Section 3.3).

We then explore the (relatively) intuitive model of sequential consistency (SC). SC is important because it is what many programmers expect of shared memory and provides a foundation for understanding the more relaxed (weak) memory consistency models presented in the next two chapters. We first present the basic idea of SC (Section 3.4) and present a formalism of it that we will also use in subsequent chapters (Section 3.5). We then discuss implementations of SC, starting with naive implementations that serve as operational models (Section 3.6), a basic implementation of SC with cache coherence (Section 3.7), more optimized implementations of SC with cache coherence (Section 3.8), and the implementation of atomic operations (Section 3.9). We conclude our discussion of SC by providing a MIPS R10000 case study (Section 3.10) and pointing to some further reading (Section 3.11).

3.1 PROBLEMS WITH SHARED MEMORY BEHAVIOR

To see why shared memory behavior must be defined, consider the example execution of two cores[1] depicted in Table 3.1 (this example, as is the case for all examples in this chapter, assumes that the initial values of all variables are zero). Most programmers would expect that core C2's register r2 should get the value NEW. Nevertheless, r2 can be 0 in some of today's computer systems.

[1] Let "core" refer to software's view of a core, which may be an actual core or a thread context of a multithreaded core.

Hardware can make r2 get the value 0 by reordering core C1's stores S1 and S2. Locally (i.e., if we look only at C1's execution and do not consider interactions with other threads), this reordering seems correct because S1 and S2 access different addresses. The sidebar on page 18 describes a few of the ways in which hardware might reorder memory accesses, including these stores. Nonhardware experts may wish to trust that such reordering can happen (e.g., with a write buffer that is not first-in–first-out).

Table 3.1: Should r2 always be set to NEW?

TABLE 3.1: Should r2 Always be Set to NEW?

Core C1	Core C2	Comments
S1: Store data = NEW;		/* Initially, data = 0 & flag ≠ SET */
S2: Store flag = SET;	L1: Load r1 = flag;	/* L1 & B1 may repeat many times */
	B1: if (r1 ≠ SET) goto L1;	
	L2: Load r2 = data;	

With the reordering of S1 and S2, the execution order may be S2, L1, L2, S1, as illustrated in Table 3.2.

Table 3.2: One possible execution of program in Table 3.1

TABLE 3.2: One Possible Execution of Program in Table 3.1.

cycle	Core C1	Core C2	Coherence state of data	Coherence state of flag
1	S2: Store flag=SET		read-only for C2	read-write for C1
2		L1: Load r1=flag	read-only for C2	read-only for C2
3		L2: Load r2=data	read-only for C2	read-only for C2
4	S1: Store data=NEW		read-write for C1	read-only for C2

Sidebar: How a Core Might Reorder Memory Access

This sidebar describes a few of the ways in which modern cores may reorder memory accesses to different addresses. Those unfamiliar with these hardware concepts may wish to skip this on first reading. Modern cores may reorder many memory accesses, but it suffices to reason about reordering two memory operations. In most cases, we need to reason only about a core reordering two

memory operations to two different addresses, as the sequential execution (i.e., von Neumann) model generally requires that operations to the same address execute in the original program order. We break the possible reorderings down into three cases based on whether the reordered memory operations are loads or stores.

Store-store reordering. Two stores may be reordered if a core has a non-FIFO write buffer that lets stores depart in a different order than the order in which they entered. This might occur if the first store misses in the cache while the second hits or if the second store can coalesce with an earlier store (i.e., before the first store). Note that these reorderings are possible even if the core executes all instructions in program order. Reordering stores to different memory addresses has no effect on a single-threaded execution. However, in the multi-threaded example of Table 3.1, reordering Core C1's stores allows Core C2 to see flag as SET before it sees the store to data. Note that the problem is not fixed even if the write buffer drains into a perfectly coherent memory hierarchy. Coherence will make all caches invisible, but the stores are already reordered.

Load-load reordering. Modern dynamically-scheduled cores may execute instructions out of program order. In the example of Table 3.1, Core C2 could execute loads L1 and L2 out of order. Considering only a single-threaded execution, this reordering seems safe because L1 and L2 are to different addresses. However, reordering Core C2's loads behaves the same as reordering Core C1's stores; if the memory references execute in the order L2, S1, S2 and L1, then r2 is assigned 0. This scenario is even more plausible if the branch statement B1 is elided, so no control dependence separates L1 and L2.

Load-storeandstore-loadreordering. Out-of-order cores may also reorder loads and stores (to different addresses) from the same thread. Reordering an earlier load with a later store (a loadstore reordering) can cause many incorrect behaviors, such as loading a value after releasing the lock that protects it (if the store is the unlock operation). The example in Table 3.3 illustrates the effect of reordering an earlier store with a later load (a store-load reordering). Reordering Core C1's accesses S1 and L1 and Core C2's accesses S2 and L2 allows the counterintuitive result that both r1 and r2 are 0. Note that store-load reorderings may also arise due to local bypassing in the commonly implemented FIFO write buffer, even with a core that executes all instructions in program order.

A reader might assume that hardware should not permit some or all of these behaviors, but without a better understanding of what behaviors are allowed, it is hard to determine a list of what hardware can and cannot do.

Table 3.3: Can both r1 and r2 be set to 0?

TABLE 3.3: Can Both r1 and r2 be Set to 0?

Core C1	Core C2	Comments
S1: x = NEW; L1: r1 = y;	S2: y = NEW; L2: r2 = x;	/* Initially, x = 0 & y = 0*/

This execution satisfies coherence because the SWMR property is not violated, so incoherence is not the underlying cause of this seemingly erroneous execution result.

Let us consider another important example inspired by Dekker's Algorithm for ensuring mutual exclusion, as depicted in Table 3.3. After execution, what values are allowed in r1 and r2? Intuitively, one might expect that there are three possibilities:

- (r1, r2) = (0, NEW) for execution S1, L1, S2, then L2
- (r1, r2) = (NEW, (0) for S2, L2, S1, and L1
- (r1, r2) = (NEW, NEW), e.g., for S1, S2, L1, and L2

Surprisingly, most real hardware, e.g., x86 systems from Intel and AMD, also allows (r1, r2) = (0, (0) because it uses first-in—first-out (FIFO) write buffers to enhance performance. As with the example in Table 3.1, all of these executions satisfy cache coherence, even (r1, r2) = (0, 0).

Some readers might object to this example because it is non-deterministic (multiple outcomes are allowed) and may be a confusing programming idiom. However, in the first place, all current multiprocessors are non-deterministic by default; all architectures of which we are aware permit multiple possible interleavings of the executions of concurrent threads. The illusion of determinism is sometimes, but not always, created by software with appropriate synchronization idioms. Thus, we must consider non-determinism when defining shared memory behavior.

Furthermore, memory behavior is usually defined for *all* executions of all programs, even those that are incorrect or intentionally subtle (e.g., for non-blocking synchronization algorithms). In Chapter 5, however, we will see some high-level language models that allow *some* executions to have undefined behavior, e.g., executions of programs with data races.

3.2 WHAT IS A MEMORY CONSISTENCY MODEL?

The examples in the last sub-section illustrate that shared memory behavior is subtle, giving value to precisely defining (a) what behaviors programmers can expect and (b) what optimizations system implementors may use. A memory consistency model disambiguates these issues.

A *memory consistency model*, or, more simply, a *memory model*, is a specification of the allowed behavior of multithreaded programs executing with shared memory. For a multithreaded program executing with specific input data, it specifies what values dynamic loads may return. Unlike a single-threaded execution, multiple correct behaviors are usually allowed, as we will see for sequential consistency (Section 3.4 and beyond).

In general, a memory consistency model MC gives rules that partition executions into those obeying MC (*MC executions*) and those disobeying MC (*non-MC executions*). This partitioning of executions, in turn, partitions implementations. An *MC implementation* is a system that permits only MC executions, while a *non-MC implementation* sometimes permits non-MC executions.

Finally, we have been vague regarding the level of programming. We begin by assuming that programs are executables in a hardware instruction set architecture, and we assume that memory accesses are to memory locations identified by physical addresses (i.e., we are not considering the impact of virtual memory and address translation). In Chapter 5, we will discuss issues with high-level languages (HLLs). We will see then, for example, that a compiler allocating a variable to a register can affect an HLL memory model in a manner similar to hardware reordering memory references.

3.3 CONSISTENCY VS. COHERENCE

Chapter 2 defined cache coherence with two invariants that we informally repeat here. The *Single-Writer—Multiple-Reader (SWMR) invariant* ensures that at any (logical) time for a memory location with a given address, either (a) one core may write (and read) the address or (b) one or more cores may only read it. The *Data-Value Invariant* ensures that updates to the memory location are passed correctly so that cached copies of the memory location always contain the most recent version.

It would seem that cache coherence defines shared memory behavior. It does not. As we can see from Figure 3.1, the coherence protocol simply provides the processor core pipeline an abstraction of a memory system. It alone cannot determine shared memory behavior—the pipeline matters too. If, for example, the pipeline reorders and presents memory operations to the coherence protocol in an order con-

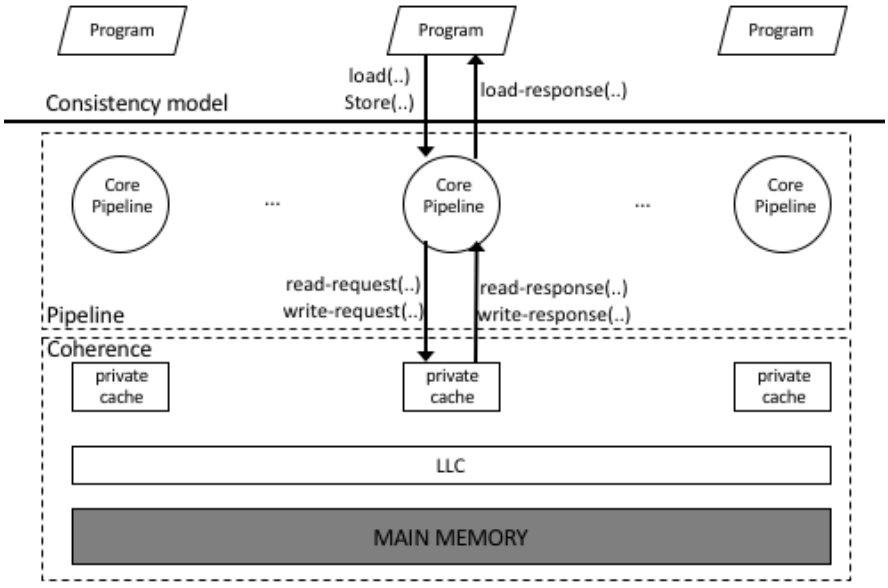

Figure 3.1: A consistency model is enforced by the processor core pipeline combined with the coherence protocol

trary to program order, (even if the coherence protocol does its job correctly) shared memory correctness may not ensue.

In summary:

- Cache coherence does not equal memory consistency.
- A memory consistency implementation can use cache coherence as a useful "black box."

3.4 BASIC IDEA OF SEQUENTIAL CONSISTENCY (SC)

Arguably the most intuitive memory consistency model is *sequential consistency (SC)*. Sequential consistency was first formalized by Lamport [11]. Lamport first called a single processor (core) *sequential* if "the result of an execution is the same as if the operations had been executed in the order specified by the program." He then called a multiprocessor *sequentially consistent* if "the result of any execution is the same as if the operations of all processors (cores) were executed in some sequential order, and the operations of each individual processor (core) appear in this sequence in the order specified by its program." This total order of operations is called *memory order*. In SC, memory order respects each core's program order, but other consis-

tency models may permit memory orders that do not always respect the program orders.

Figure 3.2 depicts an execution of the example program from Table 3.1. The middle vertical downward arrow represents the memory order (<m) while each core's downward arrow represents its program order (<p). We denote memory order using the operator <m, so op1 <m op2 implies that op1 precedes op2 in memory order. Similarly, we use the operator <p to denote program order for a given core, so op1 <p op2 implies that op1 precedes op2 in that core's program order. Under SC, memory order *respects* each core's program order. "Respects" means that op1 <p op2 implies op1 < m op2. The values in comments (/* ... */) give the value loaded or stored. This execution terminates with r2 being NEW. More generally, all executions of Table 3.1's program terminate with r2 as NEW. The only non-determinism—how many times L1 loads flag as 0 before it loads the value SET once—is unimportant.

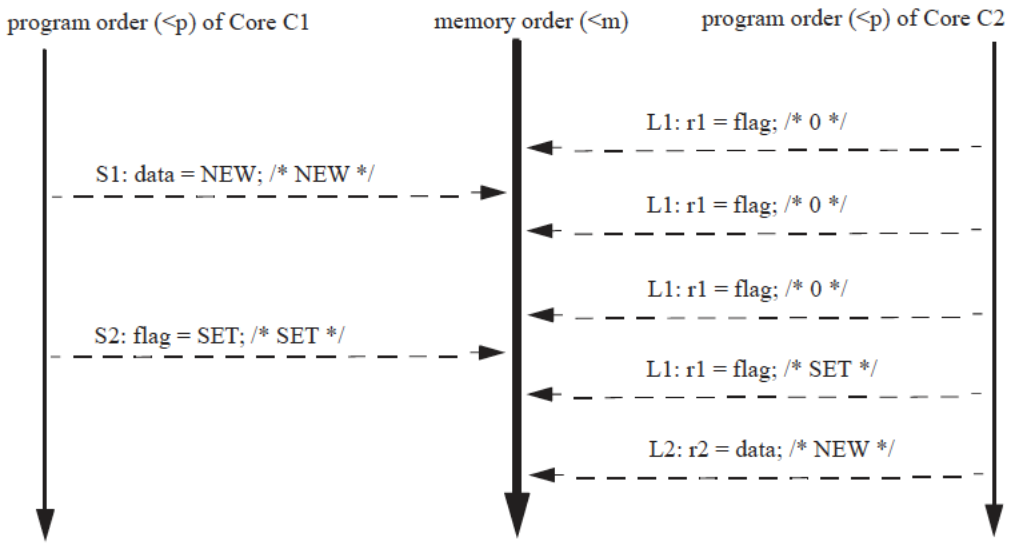

Figure 3.2: A sequentially consistent execution of Table 3.1's program.

This example illustrates the value of SC. In Section 3.1, if you expected that r2 must be NEW, you were perhaps independently inventing SC, albeit less precisely than Lamport.

The value of SC is further revealed in Figure 3.3, which illustrates four executions of the program from Table 3.3. Figure 3.3a–c depict SC executions that correspond to the three intuitive outputs: (r1, r2) = (0, NEW), (NEW, 0), or (NEW, NEW). Note that Figure 3.3c depicts only one of the four possible SC executions that leads to (r1, r2) = (NEW, NEW); this execution is {S1, S2, L1, L2}, and the others are {S1,

S2, L2, L1}, {S2, S1, L1, L2}, and {S2, S1, L2, L1}. Thus, across Figure 3.3a–c, there are six legal SC executions.

Figure 3.3d shows a non-SC execution corresponding to the output (r1, r2) = (0, 0). For this output, there is no way to create a memory order that respects program orders. Program order dictates that:

- S1 <p L1
- S2 <p L2

But memory order dictates that:

- L1 <m S2 (so r1 is (0)
- L2 <m S1 (so r2 is (0)

Honoring all these constraints results in a cycle, which is inconsistent with a total order. The extra arcs in Figure 3.3d illustrate the cycle.

We have just seen six SC executions and one non-SC execution. This can help us understand SC implementations: an SC implementation must allow one or more of the first six executions, but cannot allow the seventh execution.

We have also just observed a key distinction between consistency and coherence. Coherence applies on a per-block basis, whereas consistency is defined across all blocks. (Forecasting ahead to Chapter 7, we will see that snooping systems ensure a total order of coherence requests across all blocks, even though coherence requires only a total order of coherence requests to each individual block. This seeming overkill is required for snooping protocols to support consistency models such as SC.)

3.5 A LITTLE SC FORMALISM

In this section, we define SC more precisely, especially to allow us to compare SC with the weaker consistency models in the next two chapters. We adopt the formalism of Weaver and Germond [20]—an axiomatic method to specify consistency (more in Chapter 11 —with the following notation: L(a) and S(a) represent a load and a store, respectively, to address a. Orders <p and <m define program and global memory order, respectively. Program order <p is a per-core total order that captures the order in which each core logically (sequentially) executes memory operations. Global memory order <m is a total order on the memory operations of all cores.

An **SC execution** requires:

(1) All cores insert their loads and stores into the order <m respecting their program order, regardless of whether they are to the same or different addresses (i.e., a=b or $a \neq b$). There are four cases:

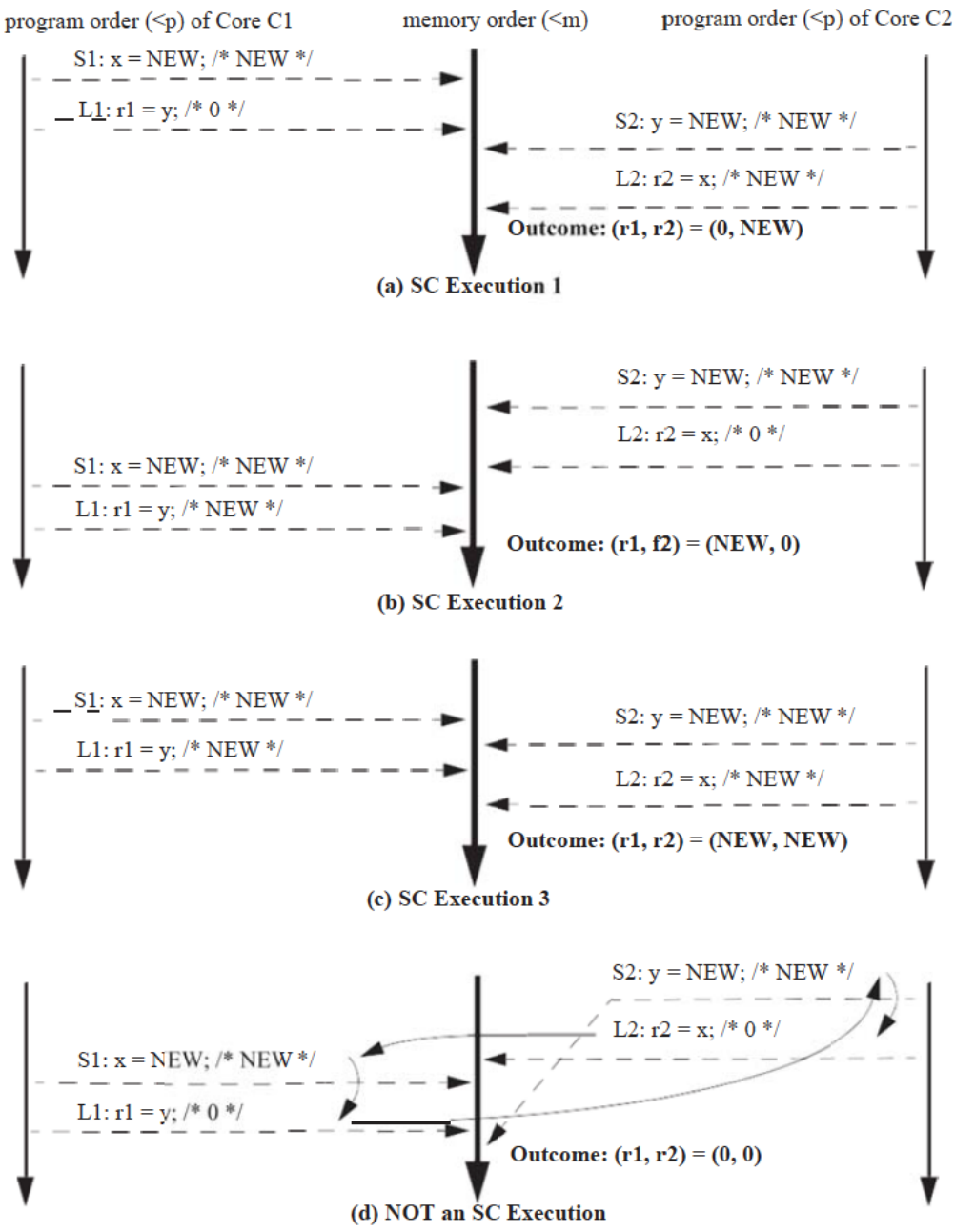

Figure 3.3: Four alternative executions of Table 3.3's program.

- If L(a) <p L(b) ⇒ L(a) <m L(b) /* Load→Load */
- If L(a) <p S(b) ⇒ L(a) <m S(b) /* Load→Store */
- If S(a) <p S(b) ⇒ S(a) <m S(b) /* Store→Store */
- If S(a) <p L(b) ⇒ S(a) <m L(b) /* Store→Load */

(2) Every load gets its value from the last store before it (in global memory order) to the same address:

Value of L(a) = Value of MAX $_{<m}$ {S(a) | S(a) <m L(a)}, where MAX $_{<m}$ denotes "latest in memory order."

Atomic read-modify-write (RMW) instructions, which we discuss in more depth in Section 3.9, further constrain allowed executions. Each execution of a test-and-set instruction, for example, requires that the load for the test and the store for the set logically appear consecutively in the memory order (i.e., no other memory operations for the same or different addresses interpose between them).

We summarize SC's ordering requirements in Table 3.4. The table specifies which program orderings are enforced by the consistency model. For example, if a given thread has a load before a store in program order (i.e., the load is "Operation 1" and the store is "Operation 2" in the table), then the table entry at this intersection is an "X" which denotes that these operations must be performed in program order. For SC, all memory operations must appear to perform in program order; under other consistency models, which we study in the next two chapters, some of these ordering constraints are relaxed (i.e., some entries in their ordering tables do not contain an "X").

An **SC implementation** permits only SC executions. Strictly speaking, this is the *safety* property for SC implementations (do no harm). SC implementations should also have some *liveness* properties (do some good). Specifically, a store must become eventually visible to a load that is repeatedly attempting to load that location. This property, referred to as eventual write-propagation, is typically ensured by the coherence protocol. More generally, starvation avoidance and some fairness are also valuable, but these issues are beyond the scope of this discussion.

3.6 NAIVE SC IMPLEMENTATIONS

SC permits two naive implementations that make it easier to understand which executions SC permits.

The Multitasking Uniprocessor

First, one can implement SC for multi-threaded user-level software by executing all threads on a single sequential core (a uniprocessor). Thread T1's instructions exe-

Table 3.4: SC ordering rules. An "X" denotes an enforced ordering.

TABLE 3.4: SC Ordering Rules. An "X" Denotes an Enforced Ordering.

		Operation 2		
		Load	Store	RMW
Operation 1	Load	X	X	X
	Store	X	X	X
	RMW	X	X	X

cute on core C1 until a context switch to thread T2, etc. On a context switch, any pending memory operations must be completed before switching to the new thread. An inspection reveals that all SC rules are obeyed.

The Switch

Second, one can implement SC with a set of cores Ci, a single switch, and memory, as depicted in Figure 3.4. Assume that each core presents memory operations to the switch one at a time in its program order. Each core can use any optimizations that do not affect the order in which it presents memory operations to the switch. For example, a simple 5-stage in-order pipeline with branch prediction can be used.

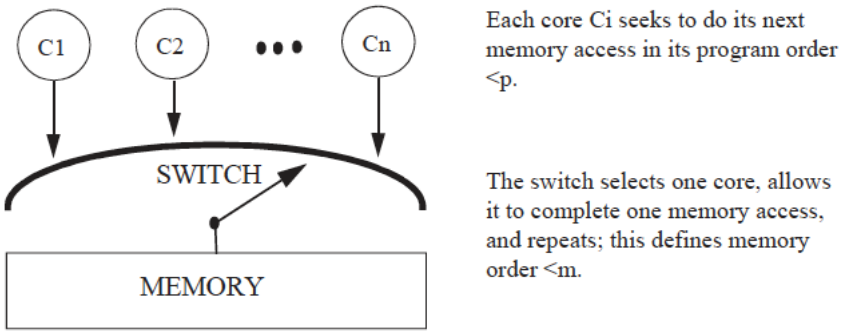

Figure 3.4: A simple SC implementation using a memory switch.

Assume next that the switch picks one core, allows memory to fully satisfy the load or store, and repeats this process as long as requests exist. The switch may pick cores by any method (e.g., random) that does not starve a core with a ready request. This implementation operationally implements SC by construction.

Assessment

The good news from these implementations is that they provide operational models defining (1) allowed SC executions and (2) SC implementation "gold standards." (In Chapter11, we will see that such operational models can be used to formally specify consistency models.) The switch implementation also serves as an existence proof that SC can be implemented without caches or coherence.

The bad news, of course, is that the performance of these implementations does not scale up with increasing core count, due to the sequential bottleneck of using a single core in the first case and the single switch/memory in the second case. These bottlenecks have led some people to incorrectly conclude that SC precludes true parallel execution. It does not, as we will see next.

3.7 A BASIC SC IMPLEMENTATION WITH CACHE COHERENCE

Cache coherence facilitates SC implementations that can execute *non-conflicting* loads and stores—two operations *conflict* if they are to the same address and at least one of them is a store—completely in parallel. Moreover, creating such a system is conceptually simple.

Here, we treat coherence as mostly a black box that implements the Single-Writer–Multiple Reader (SWMR) invariant of Chapter 2. We provide some implementation intuition by opening the coherence block box slightly to reveal simple level-one (L1) caches that:

- use state *modified (M)* to denote an L1 block that one core can write and read,
- use state *shared (S)* to denote an L1 block that one or more cores can only read, and
- have *GetM* and *GetS* denote coherence requests to obtain a block in M and S, respectively.

We do not require a deep understanding of how coherence is implemented, as discussed in Chapter 6 and beyond.

Figure 3.5a depicts the model of Figure 3.4 with the switch and memory replaced by a cache-coherent memory system represented as a black box. Each core presents memory operations to the cache-coherent memory system one at a time in

its program order. The memory system fully satisfies each request before beginning the next request for the same core.

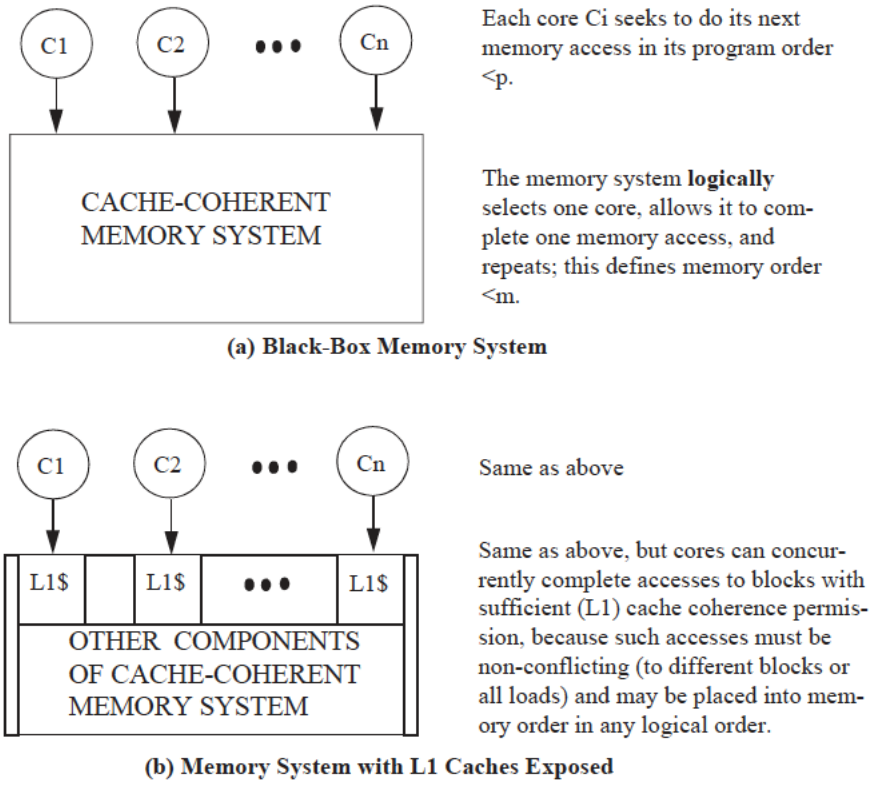

Figure 3.5: Implementing SC with cache coherence.

Figure 3.5b "opens" the memory system black box a little to reveal that each core connects to its own L1 cache (we will talk about multithreading later). The memory system can respond to a load or store to block B if it has B with appropriate coherence permissions (state M or S for loads and M for stores). Moreover, the memory system can respond to requests from different cores in parallel, provided that the corresponding L1 caches have the appropriate permissions. For example, Figure 3.6a depicts the cache states before four cores each seek to do a memory operation. The four operations do not conflict, can be satisfied by their respective L1 caches, and therefore can be done concurrently. As depicted in Figure 3.6b, we can arbitrarily order these operations to obtain a legal SC execution model. More generally, operations that can be satisfied by L1 caches always can be done concur-

rently because coherence's single-writer–multiple-reader invariant ensures they are non-conflicting.

Assessment

We have created an implementation of SC that:

- fully exploits the latency and bandwidth benefits of caches,
- is as scalable as the cache coherence protocol it uses, and
- decouples the complexities of implementing cores from implementing coherence.

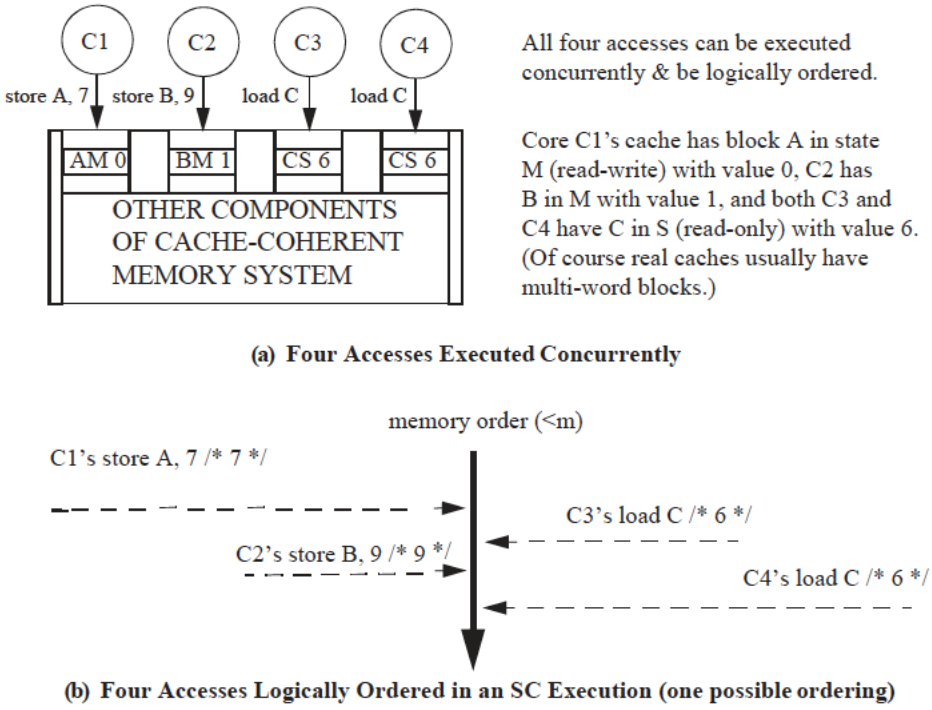

Figure 3.6: A concurrent SC execution with cache coherence.

3.8 OPTIMIZED SC IMPLEMENTATIONS WITH CACHE COHERENCE

Most real core implementations are more complicated than our basic SC implementation with cache coherence. Cores employ features like prefetching, speculative ex-

ecution, and multithreading in order to improve performance and tolerate memory access latencies. These features interact with the memory interface, and we now discuss how these features impact the implementation of SC.

Non-Binding Prefetching

A non-binding prefetch for block B is a request to the coherent memory system to change B's coherence state in one or more caches. Most commonly, prefetches are requested by software, core hardware, or the cache hardware to change B's state in the level-one cache to permit loads (e.g., B's state is M or S) or loads and stores (B's state is M) by issuing coherence requests such as GetS and GetM. Importantly, in no case does a non-binding prefetch change the state of a register or data in block B. The effect of the non-binding prefetch is limited to within the "cache-coherent memory system" block of Figure 3.5, making the effect of non-binding prefetches on the memory consistency model to be the functional equivalent of a no-op. So long as the loads and stores are performed in program order, it does not matter in what order coherence permissions are obtained.

Implementations may do non-binding prefetches without affecting the memory consistency model. This is useful for both internal cache prefetching (e.g., stream buffers) and more aggressive cores.

Speculative Cores

Consider a core that executes instructions in program order, but also does branch prediction wherein subsequent instructions, including loads and stores, begin execution, but may be squashed (i.e., have their effects nullified) on a branch misprediction. These squashed loads and stores can be made to look like non-binding prefetches, enabling this speculation to be correct because it has no effect on SC. A load after a branch prediction can be presented to the L1 cache, wherein it either misses (causing a non-binding GetS prefetch) or hits and then returns a value to a register. If the load is squashed, the core discards the register update, erasing any functional effect from the load—as if it never happened. The cache does not undo non-binding prefetches, as doing so is not necessary and prefetching the block can help performance if the load gets re-executed. For stores, the core may issue a non-binding GetM prefetch early, but it does not present the store to the cache until the store is guaranteed to commit.

Flashback to Quiz Question 1: In a system that maintains sequential consistency, a core must issue coherence requests in program order. *True or false?*
Answer: *False!* A core may issue coherence requests in any order.

Dynamically Scheduled Cores

Many modern cores dynamically schedule instruction execution out of program order to achieve greater performance than statically scheduled cores that must execute instructions in strict program order. A single-core processor that uses dynamic or out-of-(program-)order scheduling must simply enforce true data dependences within the program. However, in the context of a multicore processor, dynamic scheduling introduces a new issue: memory consistency speculation. Consider a core that wishes to dynamically reorder the execution of two loads, L1 and L2 (e.g., because L2's address is computed before L1's address). Many cores will speculatively execute L2 before L1, and they are predicting that this reordering is not visible to other cores, which would violate SC.

Speculating on SC requires that the core verify that the prediction is correct. Gharachorloo et al. [7] presented two techniques for performing this check. First, after the core speculatively executes L2, but before it commits L2, the core could check that the speculatively accessed block has not left the cache. So long as the block remains in the cache, its value could not have changed between the load's execution and its commit. To perform this check, the core tracks the address loaded by L2 and compares it to blocks evicted and to incoming coherence requests. An incoming GetM indicates that another core could observe L2 out of order, and this GetM would imply a mis-speculation and squash the speculative execution.

The second checking technique is to replay each speculative load when the core is ready to commit the load[2] [2, 17]. If the value loaded at commit does not equal the value that was previously loaded speculatively, then the prediction was incorrect. In the example, if the replayed load value of L2 is not the same as the originally loaded value of L2, then the load—load reordering has resulted in an observably different execution and the speculative execution must be squashed.

Non-Binding Prefetching in Dynamically Scheduled Cores

A dynamically scheduled core is likely to encounter load and store misses out of program order. For example, assume that program order is Load A, Store B, then Store C. The core may initiate non-binding prefetches "out of order," e.g., GetM C first and then GetS A and GetM B in parallel. SC is not affected by the order of non-binding prefetches. SC requires only that a core's loads and stores (appear to) access its level-one cache in program order. Coherence requires the level-one cache blocks to be in the appropriate states to receive loads and stores.

Importantly, SC (or any other memory consistency model):

- dictates the order in which loads and stores (appear to) get applied to coherent memory but

[2] Roth [17] demonstrated a scheme for avoiding many load replays by determining when they are not necessary.

- does NOT dictate the order of coherence activity.

Flashback to Quiz Question 2: The memory consistency model specifies the legar orderings of coherence transations. *True or false?*
Answer: *False!*

Multithreading

Multithreading—at coarse grain, fine grain, or simultaneous—can be accommodated by SC implementations. Each multithreaded core should be made logically equivalent to multiple (virtual) cores sharing each level-one cache via a switch where the cache chooses which virtual core to service next. Moreover, each cache can actually serve multiple non-conflicting requests concurrently because it can pretend that they were serviced in some order. One challenge is ensuring that a thread T1 cannot read a value written by another thread T2 on the same core before the store has been made "visible" to threads on other cores. Thus, while thread T1 may read the value as soon as thread T2 inserts the store in the memory order (e.g., by writing it to a cache block in state M), it cannot read the value from a shared load-store queue in the processor core.

3.9 ATOMIC OPERATIONS WITH SC

To write multithreaded code, a programmer needs to be able to synchronize the threads, and such synchronization often involves atomically performing pairs of operations. This functionality is pro-vided by instructions that atomically perform a "read–modify–write" (RMW), such as the well-known "test-and-set," "fetch-and-increment," and "compare-and-swap." These atomic instructions are critical for proper synchronization and are used to implement spin-locks and other synchronization primitives. For a spin-lock, a programmer might use an RMW to atomically read whether the lock's value is unlocked (e.g., equal to (0) and write the locked value (e.g., equal to 1). For the RMW to be atomic, the read (load) and write (store) operations of the RMW must appear consecutively in the total order of operations required by SC.

Implementing atomic instructions in the microarchitecture is conceptually straightforward, but naive designs can lead to poor performance for atomic instructions. A correct but simplistic approach to implementing atomic instructions would be for the core to effectively lock the memory system (i.e., prevent other cores from issuing memory accesses) and perform its read, modify, and write operations to memory. This implementation, although correct and intuitive, sacrifices performance.

More aggressive implementations of RMWs leverage the insight that SC requires only the appearance of a total order of all requests. Thus, an atomic RMW can be implemented by first having a core obtain the block in state M in its cache, if the block is not already there in that state. The core then needs to only load and store the block in its cache—without any coherence messages or bus locking—as long as it waits to service any incoming coherence request for the block until after the store. This waiting does not risk deadlock because the store is guaranteed to complete.

Flashback to Quiz 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?*
Answer: *False!*

An even more optimized implementation of RMWs could allow more time between when the load part and store part perform, without violating atomicity. Consider the case where the block is in a read-only state in the cache. The load part of the RMW can speculatively perform immediately, while the cache controller issues a coherence request to upgrade the block's state to read-write. When the block is then obtained in read-write state, the write part of the RMW performs. As long as the core can maintain the illusion of atomicity, this implementation is correct. To check whether the illusion of atomicity is maintained, the core must check whether the loaded block is evicted from the cache between the load part and the store part; this speculation support is the same as that needed for detecting mis-speculation in SC (Section 3.8).

3.10 PUTTING IT ALL TOGETHER: MIPS R10000

The MIPS R10000 [21] provides a venerable, but clean, commercial example for a speculative microprocessor that implements SC in cooperation with a cache-coherent memory hierarchy. Herein, we concentrate on aspects of the R10000 that pertain to implementing memory consistency.

The R10000 is a four-way superscalar RISC processor core with branch prediction and out-of-order execution. The chip supports writeback caches for L1 instructions and L1 data, as well as a private interface to an (off-chip) unified L2 cache.

The chip's main *system interface bus* supports cache coherence for up to four processors, as depicted in Figure 3.7 (adapted from Figure 1 in Yeager [21]). To construct an R10000-based system with more processors, such as the SGI Origin 2000 (discussed at length in Section 8.8.1), architects implemented a directory coherence protocol that connects R10000 processors via the system interface bus and a special-

ized Hub chip. In both cases, the R10000 processor core sees a coherent memory system that happens to be partially on-chip and partially off-chip.

During execution, an R10000 core issues (speculative) loads and stores in program order into an *address queue*. A load obtains a (speculative) value from the last store before it to the same address or, if none, the data cache. Loads and stores commit in program order and then remove their address queue entries. To commit a store, the L1 cache must hold the block in state M and the store's value must be written atomically with the commit.

Importantly, the eviction of a cache block—due to a coherence invalidation or to make room for another block—that contains a load's address in the address queue squashes the load and all subsequent instructions, which then re-execute. Thus, when a load finally commits, the loaded block was continuously in the cache between when it executed and when it commits, so it must get the same value as if it executed at commit. Because stores actually write to the cache at commit, the R10000 logically presents loads and stores in program order to the coherent memory system, thereby implementing SC, as discussed above.

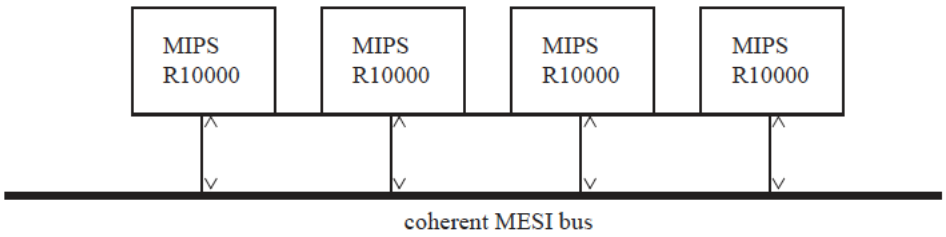

Figure 3.7: Coherent MESI bus connects up to four MIPS R10000 processors.

3.11 FURTHER READING REGARDING SC

Below we highlight a few of the papers from the vast literature surrounding SC.

Lamport [11] defined SC. As far as we know, Meixner and Sorin [14, 15] were the first to prove that a system in which cores present loads and stores in program order to a cache coherent memory system was sufficient to implement SC, even as this result was intuitively believed for some time.

SC can be compared with database serializability [9]. The two concepts are similar in that they both insist that the operations from all entities appear to affect shared state in a serial order. The concepts differ due to the nature of and expectation for operations and shared state. With SC, each operation is a single memory access to volatile state (memory) that is assumed not to fail. With serializability, each operation is a transaction on a database that can read and write multiple data-base

entities and is expected to obey ACID properties: Atomic—all or nothing even with failures, Consistent—leave the database consistent, Isolated—no effect from concurrent transactions, and Durable—effects survive crashes and power loss.

We followed Lamport and SPARC to define a total order of all memory accesses. While this can ease intuition for some, it is not necessary. Let two accesses *conflict* if they are from different threads, access the same location, and at least one is a store (or read-modify-write). Instead of a total order, one can just define the constraints on conflicting accesses and leave non-conflicting accesses unordered, as pioneered by Shasha and Snir [18]. This view can be especially valuable for the relaxed models of Chapter 5.

There have been many papers on aggressive implementations of SC. Gharachorloo et al. [7] show that non-binding prefetches and speculative execution are permitted when implementing SC and other memory models. Ranganathan et al. [16] and Gniady et al. [8] seek to speculatively retire (commit) instructions (freeing resources) and handle SC violations with secondary mechanisms. Recent work has implemented SC by building on implicit transactions and related mechanisms [1, 3, 10, 19]. All of these works use speculation and prefetching to execute memory operations out of order while completing them in program order. But there have been a couple of papers that have shown that it is possible to even non-speculatively complete memory operations out of order while enforcing SC, if the reordering is not visible to other cores. For example, two accesses to thread-private variables can be safely reordered. Whereas Singh et al. [5] seek the help of static compiler and memory management unit to determine such safe accesses, Lin et al [6] and Gope and Lipasti [4] employ hardware support for this.

Finally, a cautionary tale. We stated earlier (Section 3.7) that one way to check whether a speculatively executed load could have been observed out of order is to remember the value A that is speculatively read by a load and to commit the load if, at commit, the memory location has the same value A. Martin et al. [13] show that this is *not* the case for cores that perform value prediction [12]. With value prediction, when a load executes, the core can speculate on its value. Consider a core that speculates that a load of block X will produce the value A, although the value is actually B. Between when the core speculates on the load of X and when it replays the load at commit, another core changes block X's value to A. The core then replays the load at commit, compares the two values, which are equal, and mistakenly determines that the speculation was correct. The system can violate SC if it speculates in this way. This situation is analogous to the so-called ABA problem (`http://en.wikipedia.org/wiki/ABA_problem`), and Martin et al. showed that there are ways of checking speculation in the presence of value prediction that avoid the possibility of consistency violations (e.g., by also replaying all loads dependent on the

initially speculated load). The point of this discussion is not to delve into the details of this particular corner case or its solutions, but rather to convince you to prove that your implementation is correct rather than rely on intuition.

REFERENCES

[1] C. Blundell, M. M. K. Martin, and T. F. Wenisch. InvisiFence: Performance-transparent memory ordering in conventional multiprocessors. In *Proc. of the 36th Annual International Symposium on Computer Architecture*, June 2009. 38

[2] H. W. Cain and M. H. Lipasti. Memory ordering: A value-based approach. In *Proc. of the 31st Annual International Symposium on Computer Architecture*, June 2004. DOI: 10.1109/ISCA.2004.1310766 34

[3] L. Ceze, J. Tuck, P. Montesinos, and J. Torrellas. BulkSC: Bulk enforcement of sequential consistency. In *Proc. of the 34th Annual International Symposium on Computer Architecture*, June 2007. 38

[4] D. Gope, M. H. Lipasti Atomic SC for Simple In-order Processors In *20th IEEE International Symposium on High Performance Computer Architecture*, 2014. 38

[5] A. Singh, S. Narayanasamy, D. Marino, T.D Millstein, M. Musuvathi End-to-end sequential consistency In *39th International Symposium on Computer Architecture*, 2012. 38

[6] C. Lin, V. Nagarajan, R. Gupta and B. Rajaram. Efficient Sequential Consistency via Conflict Ordering. In *Proceedings of the 17th International Conference on Architectural Support for Programming Languages and Operating Systems*, *ASPLOS*, 2012. 38

[7] K. Gharachorloo, A. Gupta, and J. Hennessy. Two techniques to enhance the performance of memory consistency models. In *Proc. of the International Conference on Parallel Processing*, vol. I, pp. 355–64, August 1991. 34, 38

[8] C. Gniady, B. Falsafi, and T. Vijaykumar. Is SC + ILP = RC? In *Proc. of the 26th Annual International Symposium on Computer Architecture*, pp. 162–71, May 1999. 38

[9] J. Gray and A. Reuter. *Transaction Processing: Concepts and Techniques*. Morgan Kaufmann Publishers, 1993. 37

[10] L. Hammond et al. Transactional memory coherence and consistency. In *Proc. of the 31st Annual International Symposium on Computer Architecture*, June 2004. 38

[11] L. Lamport. How to make a multiprocessor computer that correctly executes multiprocess programs. *IEEE Transactions on Computers*, C-28(9):690–91, September 1979. DOI: 10.1109/TC.1979.1675439 24, 37

[12] M. H. Lipasti and J. P. Shen. Exceeding the dataflow limit via value prediction. In *Proc. of the 29th Annual IEEE/ACM International Symposium on Microarchitecture*, pp. 226–37, December 1996. DOI: 10.1109/MICRO.1996.566464 38

[13] M. M. K. Martin, D. J. Sorin, H. W. Cain, M. D. Hill, and M. H. Lipasti. Correctly implementing value prediction in microprocessors that support multithreading or multiprocessing. In *Proc. of the 34th Annual IEEE/ACM International Symposium on Microarchitecture*, pp. 328–37, December 2001. DOI: 10.1109/MICRO.2001.991130 38

[14] A. Meixner and D. J. Sorin. Dynamic verification of memory consistency in cache-coherent multithreaded computer architectures. In *Proc. of the International Conference on Dependable Systems and Networks*, pp. 73–82, June 2006. DOI: 10.1109/DSN.2006.29 37

[15] A. Meixner and D. J. Sorin. Dynamic verification of memory consistency in cache-coherent multithreaded computer architectures. *IEEE Transactions on Dependable and Secure Computing*, 6(1):282–312, 2009. 37

[16] P. Ranganathan, V. S. Pai, and S. V. Adve. Using speculative retirement and larger instruction windows to narrow the performance gap between memory consistency models. In *Proc. of the 9th ACM Symposium on Parallel Algorithms and Architectures*, pp. 199–210, June 1997. DOI: 10.1145/258492.258512 38

[17] A. Roth. Store vulnerability window (SVW): Re-execution filtering for enhanced load optimization. In *Proc. of the 32nd Annual International Symposium on Computer Architecture*, June 2005. DOI: 10.1109/ISCA.2005.48 34

[18] D. Shasha and M. Snir. Efficient and correct execution of parallel programs that share memory. *ACM Transactions on Programming Languages and Systems*, 10(2):282–312, April 1988. DOI: 10.1145/42190.42277 38

[19] T. F. Wenisch, A. Ailamaki, A. Moshovos, and B. Falsafi. Mechanisms for store-wait-free multiprocessors. In *Proc. of the 34th Annual International Symposium on Computer Architecture*, June 2007. 38

[20] D. L. Weaver and T. Germond, Eds. *SPARC Architecture Manual (Version 9)*. PTR Prentice Hall, 1994. 26

[21] K. C. Yeager. The MIPS R10000 superscalar microprocessor. *IEEE Micro*, 16(2):28–40, April 1996. DOI: 10.1109/40.491460 36