9

Multicore Systems: Coherence, Consistency, and Transactional Memory

In the preceding chapters on the design of caches, and the design of the on-chip network, we have been introduced to the incredibly complex and intricate nature of cache design. The chip is a sea of cores, cache banks, and network elements. Moreover, a cache is no more a simple matrix of memory cells. It is rather a complex structure that can be distributed all over the chip. It does not have a homogeneous access latency. Instead, the access latency is dominated by the latency of the routers' pipelines and wire delays. We need to have an elaborate on-chip network to route messages to the desired cache bank, which can be at the opposite end of the chip. Additionally, blocks migrate between different cache banks in NUCA caches such that we can increase the proximity between the cache block and the requesting core. To make matters more complicated, we have at least three levels of caches in a modern server processor (L1, L2, and L3) and we also have MSHRs (miss status handling registers) at each level. Just the task of locating a block can be fairly difficult in modern memory systems, because we need to search through many memory structures and send a lot of messages to different units on the chip. Instead of a simple matrix of cells, an on-chip memory system looks like a busy city with a maze of roads, where we can draw an analogy between the hundreds of cars, and memory request messages.

Writing a parallel program in such an environment with multiple cores is difficult. Recall that a *core* is defined as a full OOO pipeline that can run a program on its own. It is often accompanied by its own L1 cache and write buffers. In a multicore system, a simplistic view of the memory space ceases to hold. The view of the memory space or rather the virtual memory space that we are used to is that the memory space is a linear array of bytes. We can read or write to any location that a program is allowed to access. This abstraction holds very well for a sequential program. However, the moment we consider a parallel program, this abstraction begins to break. This is because as we have argued, the memory system is a complex microcosm of links, buffers, routers, and caches. Memory operations have variable latencies, and it is possible that the same memory operation might be visible to different cores at different points in time depending on where they are placed on the chip. For example, in modern memory systems, it is possible that if core 1 writes to a given memory address, core 2 might see the write earlier than core 3 because of the relative proximity to core 1. As we shall see in this chapter, this can lead to extremely non-intuitive behavior. There is thus a need to understand all such issues that can arise in a multicore system, and create a set of standards and specifications that both software and hardware must adhere to.

Definition 62
A core *is defined as a full OOO pipeline that has the capability to independently fetch instructions and run a program. A chip with multiple cores is known as a* multicore chip *or a* multicore processor*.*

The organization of this chapter is as follows. We shall first understand the different ways to write parallel programs in Section 9.1. In specific, we shall look at the two most common paradigms: shared memory and message passing. Once we have understood how parallel programs are written, we will appreciate the fact that even defining what it means for a program to execute correctly on a multicore system is very tough. The same program can produce multiple results or outcomes across runs – some of these may be non-intuitive (described in Section 9.2).

It is thus necessary to create a theoretical foundation of parallel computing and explain the notion of memory models. A *memory model* specifies the rules for determining the valid outcomes of a parallel program on a given machine. This will be described in Section 9.3. We shall further split our discussion into two parts: the rules for specifying the valid outcomes while considering accesses to only a single variable, and similar rules for multi-variable code sequences. The former is called *coherence* and the latter is called *memory consistency*.

To create a high-performing multicore system, it is necessary to associate a small, private L1 cache and possibly an L2 cache with each core. However, this design choice will break the notion of a unified memory system, unless we make it behave in that manner. We shall observe that if an ensemble of small caches obeys the axioms of coherence, it will behave as a large, unified cache (described in Section 9.4). This will allow us to improve the latency and bandwidth of the memory system significantly without compromising on correctness. On similar lines, we shall describe different types of memory models in Section 9.5. There is a trade-off between the types of behaviors a memory model allows and performance. We shall appreciate such issues in this section.

We shall subsequently look at the phenomenon of data races in Section 9.6: a data race is a potential bug in parallel programs that typically is avoided with the programmers' assistance. Along with discussing advancements in hardware, we shall discuss concomitant advances in programming languages for writing such programs. We shall look at one such novel paradigm called transactional memory in Section 9.7 and look at two approaches: one purely in software and one that requires some hardware support.

9.1 Parallel Programming

Let us now explain the methods of programming multiprocessors. For ease of explanation, let us draw an analogy here. Consider a group of workers in a factory. They cooperatively perform a task by communicating with each other orally. A supervisor often issues commands to the group of workers, and then they perform their work. If there is a problem, a worker indicates it by raising an alarm. Immediately, other workers rush to his assistance. In this small and simple setting, all the workers can hear each other, and see each other's actions. This proximity enables them to accomplish complex tasks.

We can alternatively consider another model, where workers cannot necessarily see or hear each other. In this case, they need to communicate with each other through a system of messages. Messages can be passed through letters, phone calls, or e-mails. In this setting, if a worker discovers a problem, he needs to send a message to the supervisor such that she can come and rectify the problem. Workers need to be typically aware of each other's identities, and explicitly send messages to all or a subset of them. It is not possible anymore to shout loudly and communicate with everybody at the same time. However, there are some advantages of this system. We can support many more workers because they do not have to be co-located. Secondly, since there are no constraints on the location of workers, they

can be located at different parts of the world and be doing very different things. This system is thus far more flexible and scalable.

Inspired by these real life scenarios, computer architects have designed a set of protocols for multiprocessors following different paradigms. The first paradigm is known as *shared memory*, where all the individual programs see the same view of the memory system. If program A sets the value of the shared variable x to 5, then program B immediately sees the change. The second setting is known as *message passing*. Here multiple programs communicate among each other by passing messages. The shared memory paradigm is more suitable for strongly coupled multiprocessors, and the message passing paradigm is more suitable for loosely coupled multiprocessors. A strongly coupled multiprocessor refers to a typical multicore system where the different programs running on different cores can share their memory space with each other, which includes their code and data. In comparison, a loosely coupled multiprocessor refers to a set of machines that are connected over the network, and do not share their code or data between each other. Note that it is possible to implement message passing on a strongly coupled multiprocessor. Likewise, it is also possible to implement an abstraction of a shared memory on an otherwise loosely coupled multiprocessor. This is known as *distributed shared memory* [Keleher et al., 1994]. However, this is typically not the norm.

9.1.1 Shared Memory

Let us try to add n numbers in parallel using a multiprocessor. The code for it is shown in Example 9. We have written the code in C++ using the OpenMP language extension.

It is easy to mistake the code for a regular sequential program, except for the directive *#pragma omp parallel*. This is the only extra semantic difference that we have added in our parallel program. It launches each iteration of this loop as a separate sub-program. Each such sub-program is known as a *thread*. A thread is defined as a sub-program that shares its address space (the heap and global variables) with other threads. It communicates with them by modifying the values of memory locations in the shared memory space. Each thread has its own set of local variables that are not accessible to other threads.

Example 9
Write a shared memory program to add a set of numbers in parallel.

Answer: *Let us assume that all the numbers are already stored in an array called numbers. It has $SIZE$ entries. Assume that the number of parallel sub-programs that can be launched is equal to N.*

```
/* variable declaration */
int partialSums[N];
int numbers[SIZE];
int result = 0;

/* initialize arrays */
...

/* parallel section */
#pragma omp parallel {
        /* get my processor id */
        int myId = omp_get_thread_num();

        /* add my portion of numbers */
        int startIdx = myId * SIZE/N;
```

```
        int endIdx = startIdx + SIZE/N;
        for(int jdx = startIdx; jdx < endIdx; jdx++)
                partialSums[myId] += numbers[jdx];
}

/* sequential section */
for(int idx=0; idx < N; idx++)
        result += partialSums[idx];
```

The number of iterations or the number of parallel threads that get launched is a system parameter that is set in advance. It is typically equal to the number of processors. In this case, it is equal to N. Thus, N copies of the parallel part of the code are launched in parallel. Each copy runs on a separate processor. Note that each of these copies of the program can access all the variables that have been declared before the invocation of the parallel section. For example, they can access *partialSums* and the *numbers* arrays. Each processor invokes the function *omp_get_thread_num*, which returns the id of the executing thread in the range $[0 \ldots (N-1)]$. Each thread uses the thread id to find the range of the array that it needs to add. It adds all the entries in the relevant portion of the array, and saves the result in its corresponding entry in the *partialSums* array. Once all the threads have completed their job, the sequential section begins. This piece of sequential code can run on any processor. This decision is made dynamically at runtime by the operating system or the parallel programming framework. To obtain the final result, it is necessary to add all the partial sums in the sequential section.

Definition 63
A thread *is a sub-program that shares its address space with other threads. It has a dedicated program counter and a local stack that it can use to define its local variables.*

A graphical representation of the computation is shown in Figure 9.1. A parent thread spawns a set of child threads. They do their own work and finally *join* when they are done. The parent thread takes over and aggregates the partial results.

There are several salient points to note here. The first is that each thread has its separate stack. A thread can use its stack to declare its local variables. Once it finishes, all the local variables in its stack are destroyed. To communicate data between the parent thread and the child threads, it is necessary to use variables that are accessible to both the threads. These variables need to be globally accessible by all the threads. The child threads can freely modify these variables and even use them to communicate with each other as well. They are additionally free to invoke the operating system, and write to external files and network devices. Once, all the threads have finished executing, they perform a *join* operation and free their state. The parent thread takes over and finishes the role of aggregating the results. Here, *join* is an example of a *synchronization operation* between threads. There can be many other types of synchronization operations between threads. The reader is referred to [Culler et al., 1998] for a detailed discussion on thread synchronization. All that the reader needs to understand is that there are a set of complicated constructs that threads can use to perform very complex tasks cooperatively. Adding a set of numbers is a very simple example. Multithreaded programs can be used to perform other complicated tasks such as matrix algebra, and even solve differential equations in parallel.

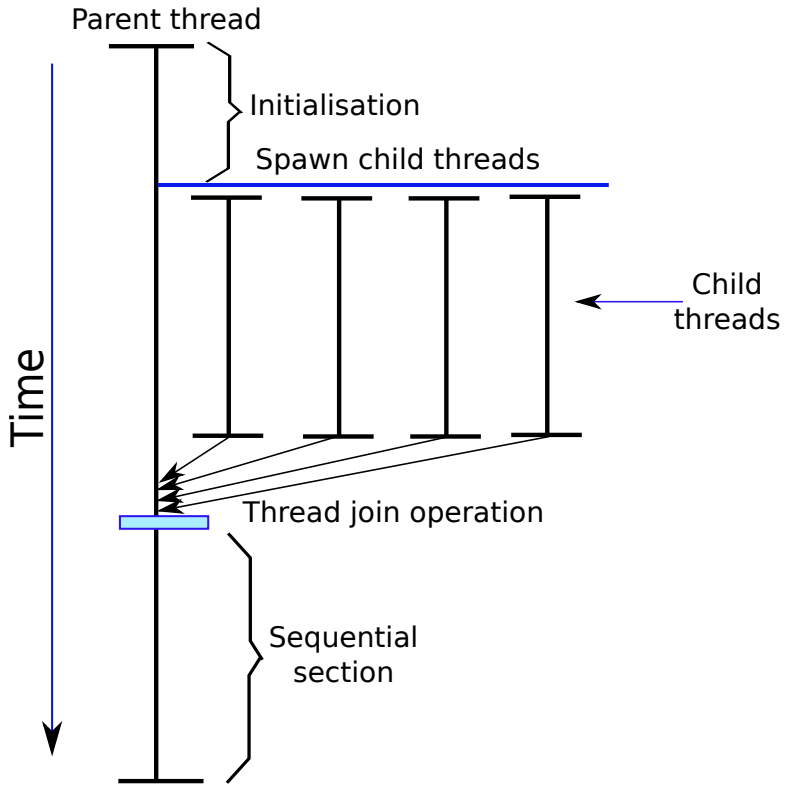

Figure 9.1: Graphical representation of the program to add numbers in parallel

9.1.2 Message Passing

Let us now briefly look at message passing. Note that message passing based loosely coupled systems are not the main focus area of this book. Hence, we shall just give the reader a flavor of message passing programs. Note that in this case, each running program is a separate entity and does not share code or data with other running programs. It is an OS *process*, where a process is defined as a running instance of a program. Typically, a process does not share its address space with any other process.

Definition 64
A process *represents the running instance of a program. Typically, it does not share its address space with any other process.*

Let us now quickly define our message passing semantics. We shall primarily use two functions *send* and *receive* as shown in Table 9.1. The $send(pid, val)$ function is used to send an integer (val) to the process whose id is equal to pid. The $receive(pid)$ is used to receive an integer sent by a process whose id is equal to pid. If pid is equal to ANYSOURCE, then the receive function can return with the value sent by any process. Our semantics is on the lines of the popular parallel programming framework MPI (Message Passing Interface) [Gropp et al., 1999]. MPI calls have many more arguments and their syntax is much more complicated than our simplistic framework. Let us now consider the same example of adding n numbers in parallel (refer to Example 10).

Function	Semantics
send (pid, val)	Send the integer *val* to the process with an id equal to *pid*.
receive (pid)	(1) Receive an integer from process pid. (2) The function blocks till it gets the value. (3) If the pid is equal to ANYSOURCE, then the *receive* function returns with the value sent by any process.

Table 9.1: *send* and *receive* calls

Example 10

Write a message passing based program to add a set of numbers in parallel. Make appropriate assumptions.

Answer: *Let us assume that all the numbers are stored in the array numbers and this array is available to all the N processors. Let the number of elements in the numbers array be SIZE. For the sake of simplicity, let us assume that SIZE is divisible by N.*

```
/* start all the parallel processes */
SpawnAllParallelProcesses();

/* For each process execute the following code */
int myId = getMyProcessId();

/* compute the partial sums */
int startIdx = myId * SIZE/N;
int endIdx = startIdx + SIZE/N;
int partialSum = 0;
for(int jdx = startIdx; jdx < endIdx; jdx++)
                partialSum += numbers[jdx];

/* All the non-root nodes send their partial sums to the root (id 0) */
if(myId != 0) {
        /* send the partial sum to the root */
        send (0, partialSum);
} else {
        /* for the root */
        int sum = partialSum;
        for (int pid = 1; pid < N; pid++) {
                sum += receive(ANYSOURCE);
        }

        /* shut down all the processes */
        shutDownAllProcesses();

        /* return the sum */
        return sum;
}
```

9.1.3 Amdahl's Law

We have now taken a look at examples for adding a set of n numbers in parallel using both the paradigms namely shared memory and message passing. We divided our program into two parts: a sequential part and a parallel part (refer to Figure 9.1). In the parallel part of the execution, each thread completed the work assigned to it and created a partial result. In the sequential part, the root or master or parent thread initialized all the variables and data structures and spawned all the child threads. After all the child threads completed (or joined), the parent thread aggregated the results produced by all the child threads. This process of aggregating results is also known as *reduction*. The process of initializing variables and reduction are both sequential.

Let us now try to derive the speedup of a parallel program vis-a-vis its sequential counterpart. Let us consider a program that takes T_{seq} units of time to execute. Let f_{seq} be the fraction of time that it spends in its sequential part and $1 - f_{seq}$ be the fraction of time that it spends in its parallel part. The sequential part is unaffected by parallelism; however, the parallel part gets equally divided among the processors. If we consider a system of P processors, then the parallel part is expected to be sped up by a factor of P. Thus, the time (T_{par}) that the parallel version of the program takes is equal to

$$T_{par} = T_{seq} \times \left(f_{seq} + \frac{1 - f_{seq}}{P} \right) \tag{9.1}$$

Alternatively, the speedup S is given by

$$S = \frac{T_{seq}}{T_{par}} = \frac{1}{f_{seq} + \frac{1-f_{seq}}{P}} \tag{9.2}$$

Equation 9.2 is known as the Amdahl's Law. It is a theoretical estimate (or rather the upper bound in most cases) of the speedup that we expect with additional parallelism.

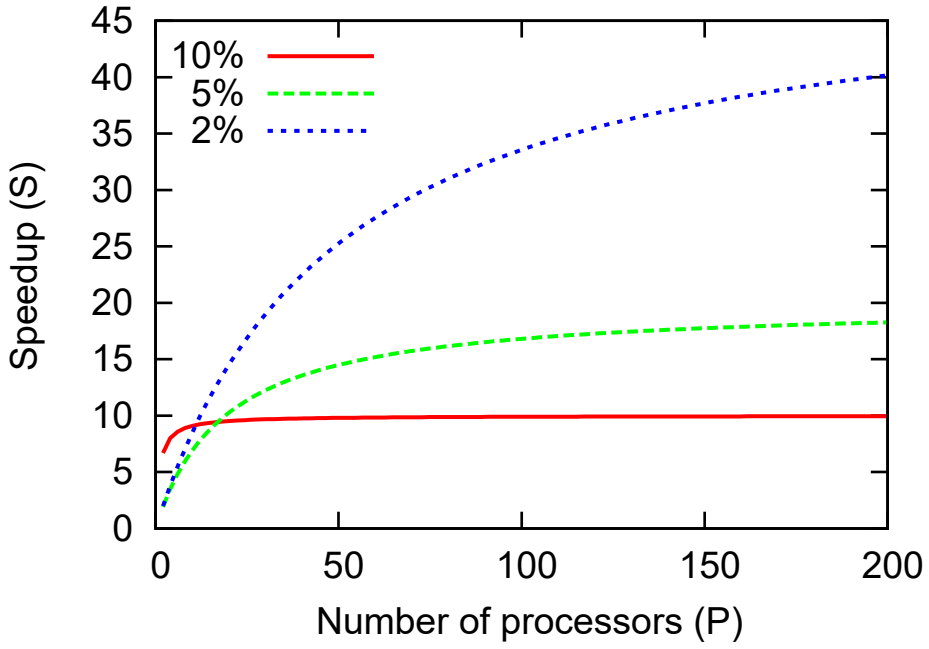

Figure 9.2: Speedup (S) vs number of processors (P)

Figure 9.2 plots the speedups as predicted by Amdahl's Law for three values of f_{seq}: 10%, 5%, and 2%. We observe that with an increasing number of processors, the speedup gradually saturates and tends to the limiting value, $1/f_{seq}$. We observe diminishing returns as we increase the number of processors beyond a certain point. For example, for $f_{seq} = 5\%$, there is no appreciable difference in speedups between a system with 35 processors and a system with 200 processors. We approach similar limits for all three values of f_{seq}. The important point to note here is that increasing speedups by adding additional processors has its limits. We cannot expect to keep getting speedups indefinitely by adding more processors because we are limited by the length of the sequential sections in programs.

To summarize, we can draw two inferences. The first is that to speed up a program it is necessary to have as much parallelism as possible. Hence, we need to have a very efficient parallel programming library and parallel hardware. However, parallelism has its limits, and it is not possible to increase the speedup appreciably beyond a certain limit. The speedup is limited by the length of the sequential section in the program. To reduce the sequential section, we need to adopt approaches both at the algorithmic level and at the system level. We need to design our algorithms in such a way that the sequential section is as short as possible. For example, in Examples 9 and 10, we can also perform the initialization in parallel (reduces the length of the sequential section). Secondly, we need a fast processor that can minimize the time it takes to execute the sequential section.

9.1.4 Gustafson-Barsis's Law

While deriving the Amdahl's law, we assumed that the size of the problem remains the same with an increasing amount of computational power. However, this is not the case in practice. When we have more resources, the problem size also increases. For example, if we are simulating an airplane wing, then we consider finer and finer meshes as we increase the number of processors. This increases the overall accuracy of the simulation.

Let the total workload be W. It has a sequential part Wf_{seq} and a parallel part $W(1 - f_{seq})$. It is the parallel part that is going to be sped up with additional processors. The speedup of the parallel part is equal to P, where P is the number of processors. We can thus do additional work in the same time. We can scale the parallel portion of the workload by a factor of P; the new workload W_{new} is as follows.

$$W_{new} = f_{seq}W + (1 - f_{seq})PW \tag{9.3}$$

The execution time on a single processor is proportional to the workload. Let the constant of proportionality be α. We thus derive the single processor execution time T_{seq} and the P-processor execution time T_{par} as follows. Note that the parallel part of the execution time gets divided by the number of processors.

$$\begin{aligned} T_{seq} &= \alpha W_{new} = \alpha(f_{seq}W + (1 - f_{seq})PW) \\ T_{par} &= \alpha(f_{seq}W + (1 - f_{seq})PW/P) = \alpha W \end{aligned} \tag{9.4}$$

The speedup S is equal to T_{seq}/T_{par}.

$$\begin{aligned} S &= \frac{T_{seq}}{T_{par}} \\ &= \frac{f_{seq}W + (1 - f_{seq})PW}{W} \\ &= f_{seq} + (1 - f_{seq})P \end{aligned} \tag{9.5}$$

Let us understand the implications of this equation. As we increase the number of processors P, the speedup increases. Ultimately $(1 - f_{seq})P$ will significantly exceed f_{seq}. Thus, the speedup for large P

will be $(1 - f_{seq})P$. This means that the only role that f_{seq} plays is in determining the slope of the curve for large P. If $f_{seq} = 0$, then the speedup is P times, which is expected when we do not have a sequential portion.

For all other values of f_{seq} where we are scaling the parallel part of the problem by a factor of P, the slope of the line is given by $(1 - f_{seq})$. Even if we are scaling the problem, we need to still limit the size of the sequential section because the absolute difference in execution times for different values of f_{seq} will be significant for large values of P.

9.1.5 Design Space of Multiprocessors

Michael J. Flynn proposed the famous Flynn's classification of multiprocessors in 1966. He started out by observing that an ensemble of different processors might either share code, data, or both. There are four possible choices – SISD (single instruction single data), SIMD (single instruction multiple data), MISD (multiple instruction single data), and MIMD (multiple instruction multiple data).

Let us describe each of these types of multiprocessors in some more detail.

SISD This is a standard uniprocessor with a single pipeline as described in Chapter 2. A SISD processor can be thought of as a special case in the universe of multiprocessors.

SIMD A SIMD processor can process multiple streams of data using a single instruction. For example, a SIMD instruction can add 4 sets of numbers with a single instruction. Modern processors incorporate SIMD instructions in their instruction set and have special SIMD execution units also. Examples include x86 processors that support the SSE and AVX instruction sets. Vector processors and, to a lesser extent, GPUs are examples of highly successful SIMD processors.

MISD MISD systems are very rare in practice. They are mostly used in systems that have very high reliability requirements. For example, large commercial aircraft typically have multiple processors running different versions of the same program/algorithm. The final outcome is decided by voting. For example, a plane might have a MIPS processor, an ARM processor, and an x86 processor, each running different versions of the same program such as an autopilot system. Here, we have multiple instruction streams, yet a single source of data. A dedicated voting circuit computes a majority vote of the three outputs. For example, it is possible that because of a bug in the program or the processor, one of the systems can erroneously take a decision to turn left. However, both of the other systems might take the correct decision to turn right. In this case, the voting circuit will decide to turn right. Since MISD systems are hardly ever used in practice, other than in such specialized situations, we shall not discuss them anymore in this book.

MIMD MIMD systems are by far the most prevalent multiprocessor systems today. Here, there are multiple instruction streams and multiple data streams. Multicore processors, and large servers are all MIMD systems. Examples 9 and 10 also showed the example of a program for a MIMD machine. We need to carefully explain the meaning of multiple instruction streams. This means that instructions come from multiple sources. Each source has its unique location and associated program counter. Two important branches of MIMD paradigms have formed over the last few years.

The first is *SPMD* (single program multiple data) and the second is *MPMD* (multiple program multiple data). Most parallel programs are written in the SPMD style (Examples 9 and 10). Here, multiple copies of the same program run on different cores or separate processors. However, each individual processing unit has a separate program counter and thus perceives a different instruction stream. Sometimes SPMD programs are written in such a way that they perform different actions depending on their thread ids. We saw a method in Example 9 on how to achieve this using OpenMP functions. The advantage of SPMD is that we do not have to write multiple programs

for different processors. Parts of the same program can run on all the processors, though their behavior might be different.

A contrasting paradigm is MPMD. Here, the programs that run on different processors or cores are actually different. They are more useful for specialized processors that have heterogeneous processing units. There is typically a single master program that assigns work to slave programs. The slave programs complete the quanta of work assigned to them and then return the results to the master program. The nature of work of both the programs is actually very different, and it is often not possible to seamlessly combine them into one single program.

From the aforementioned description, it is clear that the systems that we need to focus on are SIMD and MIMD. MISD systems are very rarely used and thus will not be discussed anymore. Let us first discuss MIMD multiprocessing. Note that we shall only describe the SPMD variant of MIMD multiprocessing because it is the most common approach.

9.1.6 Multithreading in Hardware

Up till now we have discussed multithreading in software, where a *thread* has been defined as a lightweight process that shares its address space with other threads. A multithreaded program contains multiple threads that run in parallel to complete a large task.

However, the concept of multithreading exists in hardware as well, albeit in a different form. We have very wide issue processors nowadays. It is often not possible to completely saturate the issue width. As a result, a lot of hardware resources get wasted. It is a better idea to run two parallel threads, which are not threads in the software sense – they need not share the address space. These threads may be separate processes.

High-performance processors typically run several hardware threads in parallel. If one thread is waiting for a value to return from main memory and the pipeline is getting stalled, during that time the other threads can execute. This increases the throughput of the entire system. We need to have a separate program counter and rename table for each thread. Furthermore, the id of the thread needs to be a part of the instruction packet to correctly enforce dependences. A simple idea called hyperthreading proposes to partition the resources such as the physical registers, instruction window entries, issue slots, functional units and the ROB equally among the threads. A more sophisticated approach referred to as simultaneous multithreading (SMT) proposes to flexibly allocate the resources depending upon runtime conditions and the priority of threads.

There are several kinds of multithreading. Let us discuss a few popular variants. We would like to reiterate the fact that we are discussing "hardware threads" here.

Coarse-grained multithreading In this case, we time-multiplex the pipeline. For k cycles we run thread 1, and for the next k cycles we run thread 2, and so on. If a given thread gets stalled because of an event that has a long latency such as an L2 miss or an I/O event, then we schedule another thread. In this case, k is of the order of tens of cycles. Scheduling another thread after stopping the execution of the current thread is known as "switching the context".

Fine-grained multithreading The idea is similar to coarse-grained multithreading; however, in this case, k has a much lower value. This approach is typically used when we can change the context quickly, and we would like to run another thread as soon as we detect an operation such as an L1 miss, which will take tens of cycles to execute. This approach is more responsive, yet it has more overheads in terms of switching the context between the threads.

Simultaneous multithreading (SMT) This is the most flexible approach, which is also the most complex. In this case, we fetch instructions from several threads in parallel. At runtime, the resources are dynamically partitioned between the threads. This automatically allows other threads to use as many resources as possible when one of the threads is stuck. As of 2020, a lot of server processors implement SMT.

9.2 Overview of Issues in Parallel Hardware

Now that we have an understanding of parallel programs, let us look at the hardware support that is required to run such parallel programs. Note that in the following discussion we need to reconcile two perspectives: the software perspective and the hardware perspective. The software perspective is from the point of view of programs, which have an ideal view of resources. In comparison, the hardware perspective is from the point of view of implementation. The overall aim is to ensure that software programs execute correctly without sacrificing performance. Before proceeding, the reader should thoroughly recapitulate the concept of a software *thread*, and the fact that different threads share parts of the virtual address space.

9.2.1 Shared and Distributed Caches

Let us consider two aspects of such a design – performance and correctness. Let us look at the L1 level. In a single core or a dual core system, we can have a single L1 cache. However, as the number of cores increases, having one large, shared L1 cache is not practical. If we have 16 cores, and we want to support two memory requests per core per cycle, then we need a 32-port L1 cache, which is impossible to fabricate. We can create a NUCA like organization as we had studied in Section 8.5; however, given the multicycle wire delays and increased chances of bank conflicts, this also is impractical. Hence, we cannot afford a large, shared L1 cache for all the cores in a multicore system.

For similar reasons, it is often difficult to afford a large L2 cache in a server-class multicore system. Since we cannot afford a shared cache because of performance issues, let us consider a design with distributed caches. In such a design, each core has a *private L1 cache*. The ensemble of L1 caches acts like one large shared L1 cache, albeit conceptually. We get the advantage of performance by having one small and fast L1 cache per core. It can be a very small and power efficient structure. It will allow us to take advantage of temporal and spatial locality.

Let us compare the designs with a shared L1 cache and a distributed L1 cache in Figure 9.3. Figure 9.3(a) shows a design with a shared L1 cache, and Figure 9.3(b) shows a design where each core has a private L1 cache. The L1 caches are connected using a shared bus.

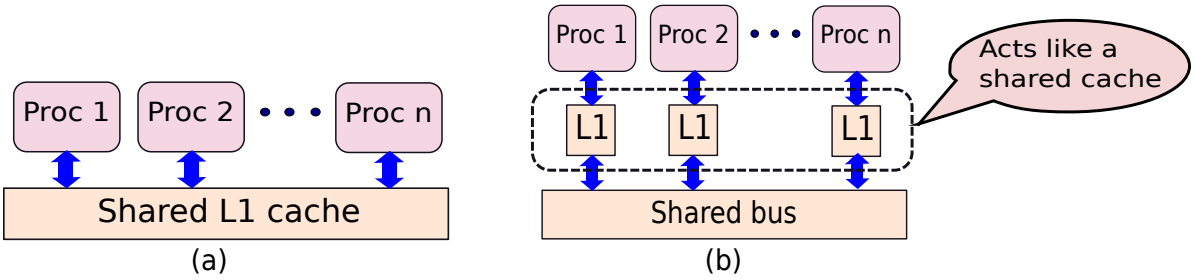

Figure 9.3: (a) Shared cache, (b) A distributed cache that conceptually acts as a single shared cache

Let us now come to the second problem – *correctness*. In a distributed cache, we need to ensure that the ensemble of L1 caches behaves as a single cache. Otherwise, the compiler needs to generate different types of code for machines that use different kinds of caches. The compiler has to be aware of the fact that the machine has a distributed L1 cache, and a write to a shared variable on one core may not be visible to the other core. This is outright impractical. Hence, to an external observer such as the programmer or the compiler, a shared and a distributed cache should look the same from the point of view of program correctness, or the outcomes of a program's execution. Ensuring the correctness of a distributed cache is known as the *cache coherence problem*. Recall that we had used a similar line of reasoning when we designed the OOO processor; we had argued that to an external observer, an OOO

processor and an in-order processor should appear to be the same (from the point of view of a program's execution).

Let us elaborate on some of the issues that we shall encounter while designing a distributed cache. Consider two cores, A and B, that are running two threads of the same application. If both the threads decide to write to variable x, then we have a problem. The memory address associated with x will be the same for both the threads. A will write to that address and keep the value in its private L1 cache. B will also do the same. If both the writes happen at more or less the same time, then we have a complex situation in our hands. We will not be in a position to find out which write operation is newer. After some time, if instructions on cores A and B start reading the value of x from memory, then they stand to read different values, even though the read operations are happening at the same time without any intervening writes. Such behaviors need to be handled properly. The reason underlying such problems is that we have two separate physical locations for the same memory address. They contain the value of the variable x. In comparison, a shared cache does not have this problem because the value of a variable (or its associated memory address) is stored in only one physical location.

The correctness issues in a distributed cache such as the one we just described, arise from the fact that for a single memory address, there are multiple locations across caches at the same level. The updates to these locations need to be somehow synchronized, otherwise this will lead to non-intuitive program behaviors. To ensure that all of these caches present a unified view of the physical address space, we need to design a *cache coherence* protocol to solve such problems. Note that all cache coherence issues and definitions are in the context of the behavior of a multithreaded program with respect to accesses to any **single memory address**. For example, in our current discussion, we only looked at all the accesses to the variable x. This definition is crucial and will be used repeatedly in later sections.

Definition 65

- *A shared cache is one large cache where we have only one physical location for a given memory address.*
- *A distributed cache comprises a group of small caches located at different places on the die. This ensemble of caches may have correctness problems because there are multiple physical locations for a given memory address.*
- *The aim is to make a distributed cache indistinguishable from a shared cache to an external observer in terms of correctness properties with regard to the outcome of memory operations. This is known as the cache coherence problem.*
- *To solve the cache coherence problem, we need a cache coherence protocol.*

Even though in most designs as of 2020, we have a shared L2 cache; however, this is not a strict necessity. We can have a private L2 cache per core, or have one private L2 cache for a group of cores. Whenever we have a distributed cache at any level, we need a cache coherence protocol.

9.2.2 Memory Consistency

Let us look at another aspect of multiprocessor memory systems that deals with accesses to multiple memory addresses. Consider the simple piece of parallel code.

Thread 1	Thread 2
x = 1;	$t1 = y$
y = 1;	$t2 = x$

x and y are global variables in a multithreaded program; assume that all our variables are initialized to zero. Here, Thread 1 is setting both x and y to 1. Thread 2 is reading y into local variable $t1$, and then is reading x into local variable $t2$.

Let us look at what is happening at the level of the NoC and cache banks. When we are updating global variables, we are essentially performing memory writes to their addresses in memory. A write to a memory address is a complex series of events. We need to create a write message and send it on the NoC to the corresponding cache bank such that the value can be written to the correct physical location. It is like sending a letter by post from New Delhi to Moscow. The writes to x and y get converted to such letters that are sent through the NoC. The same is true for a read request to memory, where the basic operation is to inject a read message into the NoC. It needs to be routed to the correct physical location, and then we need to send the value that was read back to the requesting core or cache.

In this complex sequence of messages, it is very well possible that the message to update x might get caught up in network congestion, and the message to update y might reach its destination earlier. This can happen for a myriad of reasons. Maybe the message to update y takes a different route that is less congested, or the cache bank that holds y is closer to the core that is issuing the write request. The net summary is that the updates to x and y need not be seen to be happening one after the other, or even in the same order by other cores. This can lead to pretty anomalous outcomes, which are clearly non-intuitive. Sadly, given the complex nature of interactions inside multicore processors, such occurrences are perfectly normal, and in fact many commercial processors allow many such behaviors.

For example, it is possible for a thread running on another core to read $y = t1 = 1$, and then read $x = t2 = 0$. Recall that the assumption is that all global variables such as x and y are initialized to zero. This observation would be very anomalous and non-intuitive indeed, because as per program order, we first update x and then y. The core that performs these updates sees them in that order. However, because of the non-deterministic nature of message delivery times in a realistic NoC such a situation is perfectly possible.

Even though the outcome $(t1, t2) = (1, 0)$ ($t1 = 1$ and $t2 = 0$) for Thread 2 looks to be plausible, it somehow manages to bother us and tell us that the outcome is not intuitive and hence undesirable. This is simply not how we want programs to behave. It appears that different cores are seeing different views of memory operations, and their perception of the relative order of memory operations is different. This was not happening in a single-threaded system, and thus we are not used to such outcomes. Writing correct parallel programs with such outcomes is going to be very difficult. Reasoning about their behavior and writing optimizing compilers that can possibly reorder memory accesses becomes even more difficult.

We thus need a theoretical framework that will allow us to reason about the possible and valid outcomes of multithreaded programs in large multicore processors with complex NoCs. We need to find ways to rein in the complexity of the behaviors of multithreaded programs on multicore systems and enforce certain policies. These policies are known as *memory consistency models* or simply *memory models*, which explicitly specify the rules for generating the valid outcomes of parallel programs. They preserve our notion of intuitiveness, provide a formal correctness framework, and simultaneously allow the programmer, compiler, and architecture to maximize performance.

Definition 66
Let us informally define a memory consistency model *as a policy that specifies the behavior of a parallel, multithreaded program. In general, a multithreaded program can produce numerous outcomes depending on the relative order of scheduling of the threads, and the behavior of memory operations. A memory consistency model restricts the set of allowed outcomes for a given multithreaded program. It is a set of rules that defines the interaction of memory instructions between each other.*

We shall take a detailed look at memory consistency models in Section 9.5 including their implementation aspects.

9.2.3 Difference between Coherence and Consistency

The difference between *coherence* and *consistency* is often not fully understood. There are conflicting definitions in literature. Hence, there is a need for a clarification. *Coherence* refers to the behavior of the memory system with respect to accesses to a single variable or memory address: informally, it provides the illusion that there is a single physical location for every memory address. If coherence is defined in the context of caches, we refer to it as cache coherence. Nevertheless, note that coherence is a general concept.

In comparison, *consistency* is literally defined as *adherence to specifications*. When we talk of memory consistency, we refer to the behavior of the memory system as a whole, and the fact that the outcome of every parallel program, defined in terms of the set of the results of all memory read operations, is *valid* as per the specifications. In other words, a *memory consistency model* or just a *memory model*, considers accesses to multiple memory locations. Moreover, we can have different kinds of memory models depending upon the underlying architecture. As of 2020, almost all popular memory models obey the rules of *coherence*.

Before an astute reader asks why coherence needs to be treated separately, and why cannot it simply be considered as a subset of the memory model, let us answer this question. When we delve into the practical aspects of coherence and consistency, we shall observe that enforcing coherence requires the maximum amount of hardware as compared to other aspects of a memory model. Hence, this is a special subset. Also, historically, these concepts have been developed somewhat independently.

Let us first provide a formal, mathematical framework to specify and analyze memory models before discussing the practical aspects. This is presented next.

9.3 Theoretical Foundations of Memory Models

Let us summarize the knowledge that we have gained up till now (see Waypoint 11).

Way Point 11

- *Creating one large shared cache for a parallel program is infeasible. It will be too large, too slow, and too inefficient in terms of power.*
- *Hence, it is a much better idea to have an ensemble of small caches. This is known as a distributed cache. The distributed cache however needs to appear to be a single, unified cache. If its behavior obeys the rules of coherence, this will be the case.*
- *Coherence is only one among several set of properties that modern architectures need to guarantee when it comes to correctly executing parallel, multithreaded programs. In general, the behavior of parallel programs on a machine needs to be specified by a memory model.*
- *The memory model treats each thread as a sequence of instructions and typically only considers the reads and writes. The* outcome *of a program is defined as the values read by all the read instructions across the threads. The memory model specifies the set of valid outcomes for a program on a given machine.*

9.3.1 Sequential and Parallel Executions

A shared cache contains a set of blocks, where each block contains a set of bytes. We normally do not access the memory at the level of bytes; instead, we divide the block into a set of memory words, where

each word is either 4 or 8 bytes. Let us assume that every variable requires a single memory word, and we only access memory at the granularity of words. We can also say that a memory location corresponds to a single memory word. We are primarily interested in the sequences of reads and writes to memory words issued by different threads, and their associated correctness properties. Let us try to model such sequences formally. We shall introduce a set of concepts that will help us in creating a mathematical model that can accurately explain the concepts of coherence and consistency.

Point of View

Consider a single shared cache. Let us place a hypothetical observer at a specific memory word that we are interested to monitor. We will see a series of read and write accesses made by possibly different threads (running on different cores). Since all of them are to the same location, we can order them sequentially. In this sequence of reads and writes, the correctness criterion is that each read operation returns the value of the latest write operation. The write might have been performed by the same thread, or a different thread running on a different core. A memory operation can be broken down into a request and a response. The request is typically issued by a core and the memory system issues the response. For a load, the response is the value, and for a store the response is typically empty, indicating that the store has completed successfully.

Let us now explain a very important concept in the design of parallel systems. It is the *point of view*. This basically captures what a hypothetical observer placed at a given location inside the memory system observes. This is her point of view.

The observer that sits on the memory location (let's say on the SRAM cells) sees a very simple view of the memory operations (see Figure 9.4). Every memory operation has three points of time associated with it: a time at which it starts (t_{start}), a time at which it completes (t_{comp}), and a time at which it ends (t_{end}). We shall use the generic term *memory operation* in our subsequent discussion – its exact definition depends on the observer. In this case, t_{start} refers to the time at which the request to start the operation arrives at the memory location, or alternatively, the time at which the operation to access the memory location starts. t_{end} refers to the point of time when all the actions with respect to the memory operation cease from the point of view of the memory location. t_{comp} refers to the time when the memory operation completes its action. This is a tricky concept, and needs to be explained in the context of reads and writes. A read operation completes when we have read the final value, and the value will not change henceforth. A write operation completes when we have written the value to the memory location. In this case, $t_{start} < t_{comp} < t_{end}$.

Let us explain with an example. Assume we have a core and a shared cache. The core issues a read request to read 4 bytes (a single memory word) from the memory location 20. The request is sent to the shared cache that has a single first-in first-out queue of memory requests as shown in Figure 9.4. Once the cache receives the request, it is enqueued in a dedicated queue. This time is t_{start}. Once the cache is free, we dequeue the head of the queue and send the address to the decoder of the SRAM array. The array access starts. Once we read the value of the SRAM cells at the sense amplifiers, we are sure that their values are stable, and will not change in the lifetime of the current operation. This is the completion time t_{comp}. Finally, when the response is written on the bus, this time can be treated as t_{end}.

To summarize, from the point of view of this observer, operations arrive sequentially, they complete their action (read or write), and then the responses are sent back. Operations never overlap. One operation finishes, and the next operation starts. This pattern is an example of a *sequential execution*, which is a basic concept in the concurrent systems' literature. Let us summarize.

1. A point of view is defined as the set of events that a hypothetical observer sees at a particular point in the memory system.

2. The observer sees a set of memory operations, where a memory operation can either be a read or a write. Note that the exact definition of a memory operation depends on the observer.

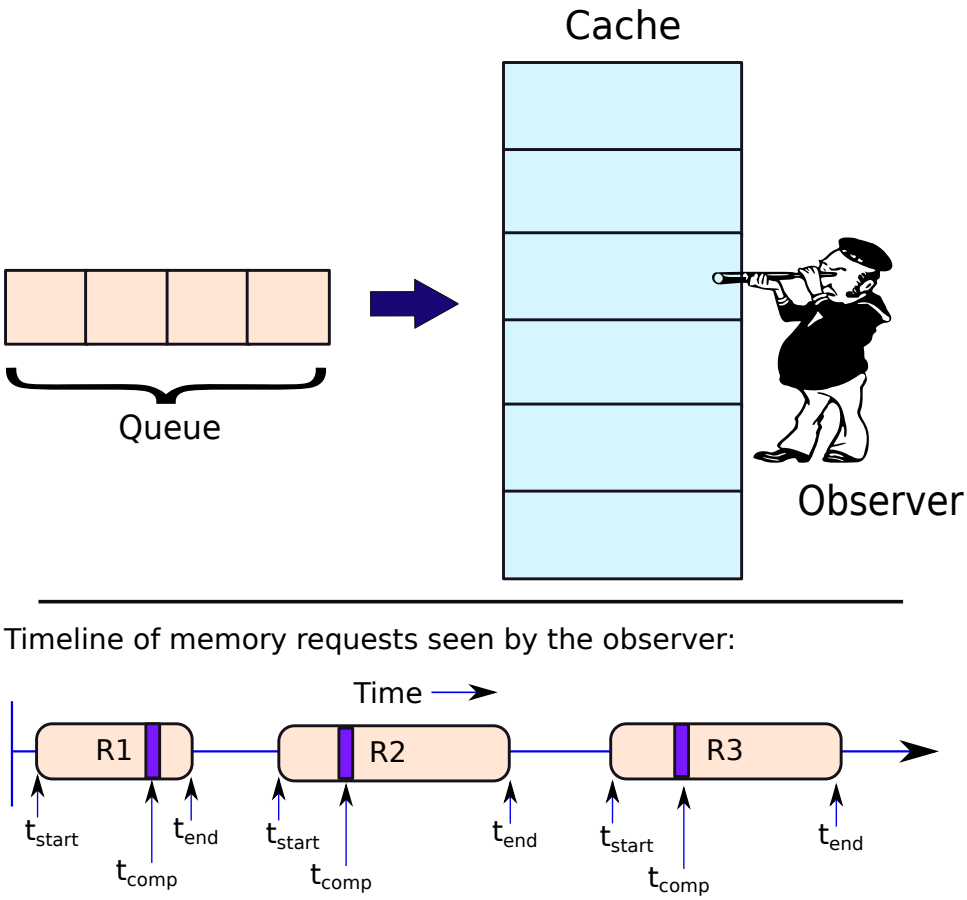

Figure 9.4: Observer at a memory location

3. Each operation has three times associated with it: t_{start}, t_{comp}, and t_{end}. The start and end times represent the times at which the observer sees the processing of the operation start and end respectively.

4. t_{comp} is the time at which the memory operation actually takes effect. For a read, it is the time, when we finally read the value, and there are no chances of the value changing for the current operation. For a write, it is the time when the value gets written, and it is potentially visible to all subsequent operations. Regardless of the observer, this relation always holds: $t_{comp} \geq t_{start}$. The relation between t_{comp} and t_{end} is slightly more tricky – it depends on the observer. Let us keep reading.

Sequential Executions

Let us recapitulate. In a shared cache, an observer sitting on a memory location sees a list of memory operations: reads or writes. Let us formally argue about what constitutes correct behavior in this case. Even though it is obvious, let us still formalize it because we will use it as a foundation for later sections.

In formal terms, an *execution* is a set of memory operations. Each operation is a 6-tuple of the form $\langle tid, t_{start}, t_{end}, type, addr, value \rangle$. *tid* is the id of the thread that has initiated the operation. t_{start} and t_{end} have been explained before. The *type* indicates if the operation is a read or a write, *addr* is the memory address, and the *value* indicates the datum that is read or written to memory. We have not

included the completion time in the definition, because it is often not known. Now, we can either have an ordered execution or a partially ordered execution.

We shall call an ordered execution a *sequential execution* where all the operations are ordered. This is not the case in partially ordered executions – there is at least one pair of operations, where an ordering between them is not specified.

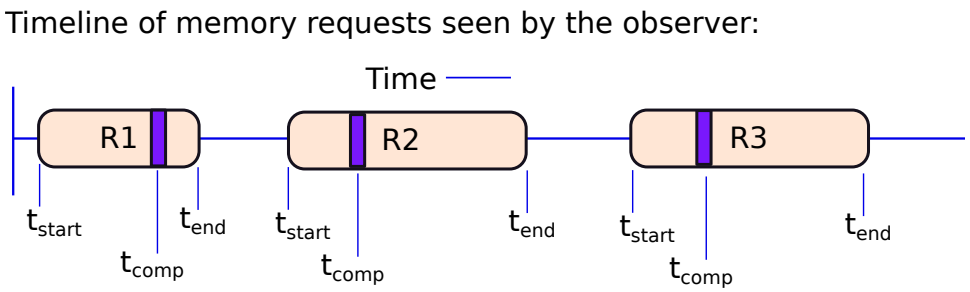

Figure 9.5: A sequential execution with three memory operations: $R1$, $R2$, and $R3$.

To understand sequential executions better, let us reproduce the relevant part of Figure 9.4 in Figure 9.5. The operations in Figure 9.5 are seen by an observer at the memory location. The end time of one operation is strictly less than the start time of the subsequent operation – they have no overlaps. This is a sequential execution where the operations are ordered by their start times (and also the completion times).

It is not necessary for operations to be non-overlapping to be part of a sequential execution. For example, we can have a pipelined cache, where before the previous operation has ended, a new operation may begin. In this case, the operations do overlap, nevertheless, there is still a sequential ordering between them – they are ordered in the ascending order of their start times.

Let us additionally define a property that establishes the *correctness* of a sequential execution. Since it consists of only read and write operations, let us take a look at all the values that are read by read operations. Each of these values needs to be correct, which means that a read operation needs to get the value of the *latest write* to the same address in the sequence. Guaranteeing that read operations get the correct values is enough because write operations do not return a value. It is only the read operations that read values from the memory system and pass them to other instructions. Let us call such a sequence where all the read operations read the correct values (latest writes) as a *legal* sequence. Also note that a legal sequence guarantees the fact that the final value of a variable is equal to the value that was last written to it. This is because if the system remains henceforth quiescent, and then we decide to read a variable a long time later, we expect to get the value of the last write.

In simple terms, a sequential execution is just an ordered sequence of memory operations. If the values that are read are correct (from the latest writes), then the execution is *legal*.

Observer at a Core

It is now clear that an observer sitting on a memory location in a shared cache observes a sequential execution, which is also legal. Let us now change the observer, and consider her point of view. Let the new observer be seated on a core that executes a single-threaded program. In this case, a memory operation from the point of view of the observer is actually a memory instruction: load or store. She can see the core executing instructions. Let us consider her point of view. For it, the *start time* is when the memory instruction is fetched, and the *end time* is when the instruction leaves the pipeline. The *completion time* is the time at which the operation actually performs its operation in the physical memory location: read or write.

In the case of a load instruction, the relation $t_{start} < t_{comp} < t_{end}$ still holds. Now, in the case of a store, the relation $t_{comp} < t_{end}$ does not necessarily hold because the value may reach the desired cache

bank much later; recall that we declare that the instruction has *ended* when it leaves the pipeline.

Let us explain this with a simple analogy. Assume that I want to send a letter. I leave my house at t_{start}, then I drop the letter in the post box at t_{end}. As far as I am concerned the operation ends when I drop the letter in the post box. Note that at that point of time, the letter has not yet reached its destination. The letter reaches the destination at the completion time t_{comp}, which happens much later. In this case, we instead have the following relation: $t_{start} < t_{end} < t_{comp}$. In fact, something similar has happened to your author once. Once he dropped a check in a drop box, and then assumed that his account has been credited a few days later when he was performing an online transaction. The transaction got declined because the check had not been picked up by the bank because of a snow storm – a real life example of t_{comp} being greater than t_{end}!

Nevertheless, in this case also the observer observes a legal sequential execution regardless of the degree of sophistication of the core. Let us prove it. Note that the order of start times (fetch times) is the same as the program order because we fetch instructions in program order. Now, we have already argued in Chapter 2 that to an external observer, the execution of an OOO processor, and an advanced in-order processor are identical (in terms of correctness) to that of a simple single-cycle processor that picks an instruction in program order, completely executes it, and then picks the next instruction. A single-cycle processor thus generates an operation stream that is a legal sequential execution – a read always gets the value of the latest write. Because the executions are identical, it means that regardless of the core, the outcome of every read operation is the same (same as the outcome in a single-cycle processor). Hence, in this case as well, we have a legal sequential execution, even though we have $t_{comp} > t_{end}$ for stores.

The implications are profound. It means that even if we speculate as much as we want, an external observer will always observe a legal sequential execution, which is a simple linear order of operations where every read gets the value written by the latest write.

Definition 67

- *An execution is a set of memory operations. Each operation is a 6-tuple of the form* $\langle tid, t_{start}, t_{end}, type, addr, value \rangle$.
- *In a sequential execution, operations are arranged in an ordered sequence. They need not be non-overlapping.*
- *A sequential execution or in general a sequence of operations is* legal, *if every read operation returns the value of the latest write operation to the same address before it in the sequence. Additionally, the final values of all the variables are equal to their last-written values.*
- *In a single-threaded program, if we order all memory operations in program order, then we arrive at a legal sequential execution.*

Parallel Executions

Up till now, we have only considered executions that have a single observer. Let us now consider a system with multiple observers. We define a *parallel execution* as follows. It extends a regular execution by also including the order of operations recorded by each observer. The set of operations recorded by the observers is mutually disjoint. Furthermore, each observer records a sequential execution. Unlike sequential executions, there is no ordering between all pair of operations. Hence, we have a partial order here. Note that in a parallel execution, unless we know the completion times at which the operations take effect, we do not know how to verify the execution. We cannot create a legal sequence.

We show one such example in Figure 9.6, where we have 3 threads that access two memory locations x and y. We have one observer per core or per thread that sees the entry and exit of instructions. The start and end times are defined in the same way as was defined for the previous example that considered a single-threaded system.

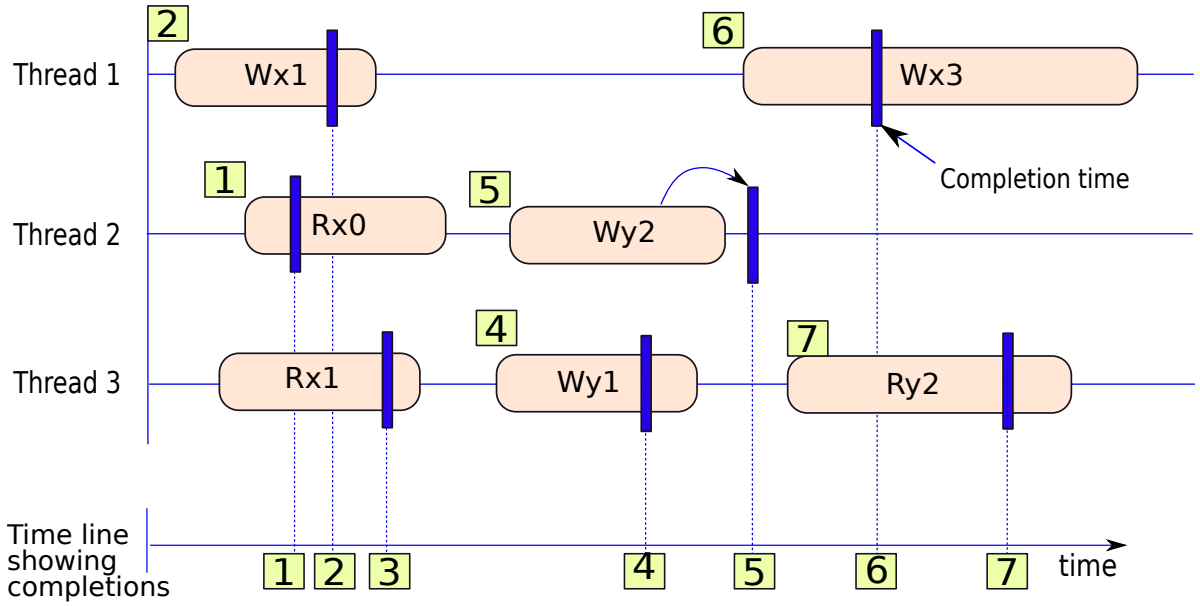

Figure 9.6: A parallel execution with 3 threads. The small vertical lines represent the completion times.

To understand this figure, let us define a standard terminology for read and write operations. Assume we are accessing the memory location corresponding to variable x. Let Rxi mean that we are reading the value i from location x. Similarly, let Wxi mean that we are writing the value i to the location x.

Let us start out by noting several interesting features of this execution. The first is that different threads running on different cores issue memory operations: reads and writes. The operations take effect based on their completion times. These times are shown with small vertical lines in the figure. Other than for $Wy2$, the rest of the completion times are between t_{start} and t_{end} in this execution. For the time being, assume that we somehow know the completion times of each operation. We shall reconsider this assumption later. Given that all the operations are ordered by their completion times, we can verify the execution.

Assume that the variables x and y are initialized to 0. In fact, we shall make this assumption for all examples henceforth – all variables stored in memory are assumed to be initialized to 0. The first instruction to complete is instruction *1* ($Rx0$). This reads the default value of x, which is 0. Subsequently, instruction *2* completes, and writes 1 to x. Then instruction *3* completes and reads $x = 1$. Let us now take a look at instructions *4* ($Wy1$) and *5* ($Wy2$). Even though they overlap, their times of completion are such that *4* completes before *5*. Hence, instruction *4*'s write gets overwritten by the write of instruction *5*. Also, note that instruction *5* is an example of an instruction where $t_{comp} > t_{end}$. Instruction *6* is the last write to x and instruction *7* reads the latest write to y ($Wy2$).

Here, each operation has a completion time, at which it appears to **take effect instantaneously**. This property is known as *atomicity*, where each memory operation appears to take effect instantaneously at its completion time. If we arrange the operations in ascending order of their completion times, then the operations appear to take effect in that order. We can thus order the operations (*1*-*7*) by their completion times in a linear timeline as shown in the bottom of the figure, and verify the correctness of

the execution.

Definition 68
A memory operation is said to be atomic *if it appears to execute instantaneously at some time t. It is pertinent to underscore the point that all the threads should perceive the fact that the operation has executed instantaneously at t. We refer to this time t as the operation's completion time.*

Problems with Parallel Executions

Unfortunately, an observer sitting on the core will not be able to perceive the completion time. She will only observe the start and end of an operation. In Figure 9.6, we assumed that we somehow know the completion time of each operation, which is impractical.

Let us ask a question to ourselves. What have we gotten by creating a parallel execution? Unlike a sequential execution, it does not allow us to arrange all the memory operations in a sequence. A sequence has some very nice properties; for example, it allows us to define a key correctness property called legality – every read returns the value of the latest write. In a parallel execution, we cannot do this. Specifying the order of operations issued by the same thread might help us visualize the execution better, but other than that, a parallel execution seems to be useless. Unlike a sequential execution, we cannot prove that it is legal or correct because the term "latest write" is not defined. The only solace is that it conveys what each observer records.

Then, why did we describe all of this discrete math to just arrive at a concept called a parallel execution that is mostly useless? The only way that we can do justice to our hard work is if we can somehow **convert** or **map** a parallel execution to a sequential execution. We know that a sequential execution is a good thing: it can be analyzed very easily, and it has some good properties like legality that is essentially a correctness property.

Equivalence between Serial and Parallel Executions

Let us introduce the notion of the *equivalence* of two executions. We will use this theoretical tool to create an equivalent sequential execution from a parallel execution.

Consider two executions P and S. Let the notation $P \mid T$ refer to all the operations in P that were issued by thread T. We similarly define $S \mid T$. This basically means that we extract all the operations from S and P that were issued by the same thread T and also preserve their order. $P \mid T$ and $S \mid T$ are thus ordered sequences. Let us use the $\equiv$ operator to denote equivalence. The task at hand is to define the conditions when $P|T \equiv S|T$.

Consider Figure 9.7. It shows the parallel execution P (originally shown in Figure 9.6) and a sequential execution S. They have the same number of operations; there is a one-to-one mapping between operation i in P and operation i' in S. Both the operations have the same type, address, thread id, and read/write the same value. They only differ in their start and end times. The start and end times of the operations belonging to S do not matter – only their relative order matters.

We say that for a given thread T, $P \mid T \equiv S \mid T$, if and only if the two ordered sequences have the same number of operations and there is a one-to-one mapping between the operations for each position in the sequences. For example, consider all the operations issued by thread 3 (T_3). $P|T_3 = \{3,4,7\}$. $S|T_3 = \{3', 4', 7'\}$. Notice the one-to-one mapping between the two sequences $P \mid T$ and $S \mid T$. Two executions P and S are said to be *equivalent*, i.e., $P \equiv S$ if for all T, $P|T \equiv S|T$. Let us quickly convince ourselves that the two executions shown in Figure 9.7 are equivalent based on our definition of equivalence.

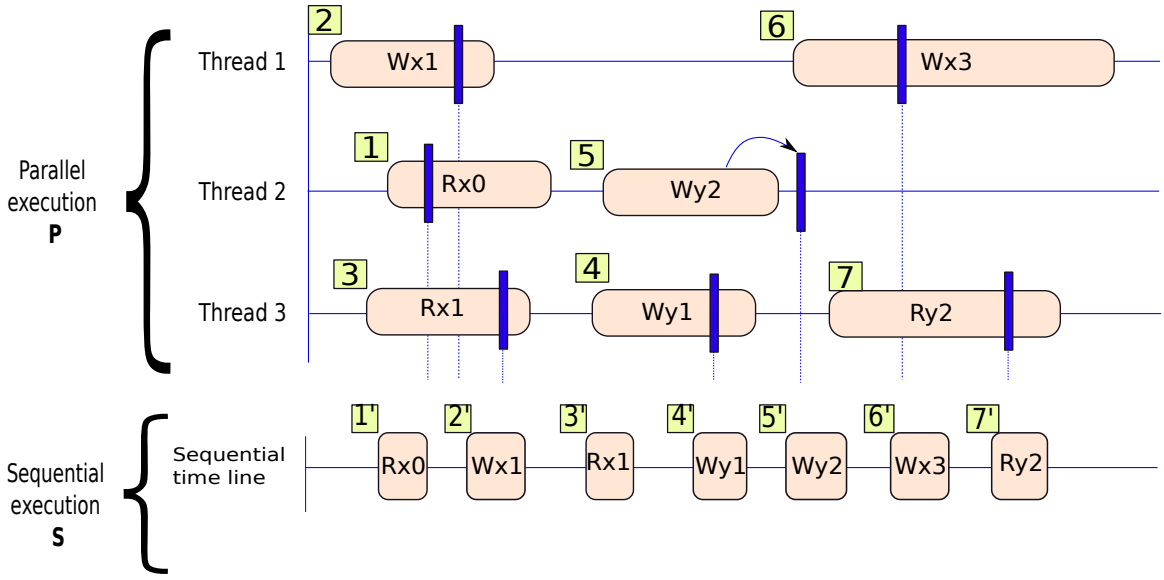

Figure 9.7: Equivalence between a sequential and parallel execution

Now note that P is a parallel execution and S is a sequential execution. The parallel execution is equivalent to a sequential execution. The readers might not have realized it yet; we have actually stumbled across one of the most effective tools in concurrency theory. We have established an equivalence between a parallel execution and a sequential execution. Hence, even if we have a parallel execution, and we do not know the completion times, there is nothing to worry. We just need to map it to an equivalent sequential execution. If the sequential execution is *legal*, then we define the parallel execution to also be *legal*. This aligns with our intuitive notion of correctness of parallel executions. The reader needs to convince herself of this fact.

Can we always map a parallel execution to a legal sequential execution? Let us find out.

9.3.2 Sequential Consistency

Consider a parallel execution where the order of operations as recorded by each observer is the same as the per-thread program order. As we saw earlier, this is the case with observers seated on cores or when the observers are the threads themselves watching their instructions get executed. This is also the default case when we are looking at the execution of a thread from the point of view of software. In such a parallel execution, each thread observes its instructions getting executed in program order.

If we can map this parallel execution to an equivalent legal sequential execution, then it means that we have a virtual ordering between all the instructions across all the threads. This is captured by the sequential execution. We can execute the instructions from the parallel threads one at a time, and arrive at the same outcomes. Even if we do not know the actual completion times, it does not matter, we can verify the equivalent sequential execution, and see if each read returns the value of the latest write to the same address.

Any such parallel execution that can be mapped to an equivalent legal sequential execution is said to be *sequentially consistent*. Sequential consistency abbreviated as *SC*, is thus a property of a parallel execution. We can however extend the definition to programs and machines. Any program that only produces sequentially consistent parallel executions or just executions is said to be sequentially consistent. We also say that the program is in SC. Similarly, if a machine only produces sequentially consistent executions, it is said to be a sequentially consistent machine.

If we think about it, sequential consistency is intuitive. The human mind always thinks sequentially,

and it is thus difficult to visualize the execution of a parallel program, and consequently argue about its correctness. However, with the notion of sequential consistency, we can do this very easily. For a parallel execution, if we can show that it is equivalent to a legal sequential execution, then we can actually think of the parallel program as a regular sequential program. We can then visualize it in our mind's eye much better and also reason about its correctness. Additionally, it is a boon to software writers particularly assembly language programmers. If the underlying architecture allows non-SC executions, it will become very difficult to write correct programs. Furthermore, programs written on one machine will not run on another. If the underlying architecture somehow guarantees only SC executions, software writers can easily write correct code that will execute seamlessly.

Definition 69
Sequential consistency has several equivalent definitions. Let us list a few popular ones.

- *If a parallel execution is equivalent to a legal sequential execution where the program order between all the operations issued by the same thread is preserved, then we say that the parallel execution is in SC.*
- *An execution is said to be sequentially consistent, or in SC, if the memory accesses of all threads can be put in a sequential order. In this sequential order, the accesses of a single thread appear in program order, and furthermore every read fetches the value of the latest write.*
- *If it is possible to interleave the memory operations of all the threads and generate a single sequence of memory operations where the operations of each thread appear in program order and the sequence is legal, then we say that the execution is sequentially consistent.*
- *Let us visualize a single-cycle processor that executes instructions from different threads by picking an instruction in program order, executing it, and writing back the results to the architectural state of the thread. We can use it to simulate the execution of parallel threads. If it is possible for it produce the same set of outcomes as a parallel execution using this sequential method of execution, then we say that the parallel execution is sequentially consistent.*

Examples: Single-variable programs

To appreciate the implications of our definitions, let us consider a few examples. First, let us consider executions that access only a single variable. These can arise out of executing multithreaded code snippets that just access a single variable, or we can extract all the accesses to a single variable from the execution of a multi-variable program.

T_1	T_2	T_3
Wx1	Rx0	Rx1
Wx2	Rx2	Rx2

Figure 9.8: SC execution

T_1	T_2	T_3
Wx1	Rx1	Rx2
Wx2	Rx2	Rx1

Figure 9.9: Execution that is not in SC

Figure 9.8 shows an SC execution (variables initialized to 0). It is possible to order the memory accesses sequentially. We can order them as follows: $Rx0 \rightarrow Wx1 \rightarrow Rx1 \rightarrow Wx2 \rightarrow Rx2$ $(T_2) \rightarrow Rx2$ (T_3). Here, the arrow $(\rightarrow)$ represents a *happens-before* relationship between operations A and B meaning that A needs to happen before B such that B can see its result.

Now consider one more execution that is not in SC in Figure 9.9. The readers need to convince themselves of this fact by trying all possible ways to create an equivalent legal sequential execution.

There is something that we fundamentally do not like in Figure 9.9. $Wx2$ comes after $Wx1$. All other threads should respect this order. However, thread T_3 does not respect it. It reads $Rx2$ before $Rx1$, which means that it sees the writes in the reverse order. This is not intuitively acceptable to us. If there are no more writes to x, threads T_2 and T_3 will have different final values of x, which breaks the notion of x being a shared variable. Hence, let us rule out this behavior – we do not like it.

One simple way to do this is to constrain **the parallel execution that corresponds to all the accesses to a single location to be sequentially consistent**. This will automatically allow the execution in Figure 9.8 and disallow the execution in Figure 9.9.

We shall discuss such issues in detail later. However, we have already started to form an opinion on what seems intuitive, and what does not. The execution in Figure 9.9 does not seem to be intuitively correct. Why do we say so? This is because it violates per-location sequential consistency (abbreviated as PLSC), which means that if we consider all the accesses to a single location (x in this case), the execution is not sequentially consistent. For now, let us assume that PLSC is a desirable property. We shall continue to look at PLSC in later sections, and keep commenting about its desirability.

Definition 70 *Consider an execution $\mathcal{E}$. For a given location x, let $\mathcal{E} \mid x$ be an execution that only contains all the accesses to x in $\mathcal{E}$. Note that we preserve the order of all memory operations. If for all x, $\mathcal{E} \mid x$ is sequentially consistent, then we say that $\mathcal{E}$ is per-location sequentially consistent or $\mathcal{E}$ is in PLSC.*

Examples: Multi-variable programs

Now let us consider the general case: executions that access multiple memory locations.

T_1	T_2
Wx1	Wy1
Ry0	Rx0

Figure 9.10: An execution that is not in SC

For the execution shown in Figure 9.10, we cannot find a sequential schedule that ensures that we read both x and y to be 0. Let us try different ways of arranging the operations:

$$Wx1 \rightarrow Wy1 \rightarrow Ry0 \rightarrow Rx0$$
$$Wx1 \rightarrow Ry0 \rightarrow Wy1 \rightarrow Rx0$$
$$Wy1 \rightarrow Rx0 \rightarrow Wx1 \rightarrow Ry0$$

All of these are **illegal** sequential executions because a read does not return the value of the latest write. The original parallel execution is thus not in SC. We interpret PLSC in this case as follows. We create two parallel executions: one for accesses to x and one for accesses to y. They are shown in Figures 9.11 and 9.12 respectively.

T_1	T_2
Wx1	
	Rx0

Figure 9.11: Accesses with respect to x

T_1	T_2
	Wy1
Ry0	

Figure 9.12: Accesses with respect to y

The reader can easily verify that both the executions are in SC. For example, the execution with respect to x is equivalent to the sequential execution $Rx0 \rightarrow Wx1$. Similarly, the execution with respect to y is equivalent to the sequential execution $Ry0 \rightarrow Wy1$. These executions thus satisfy PLSC. There is an important learning for us here.

PLSC does not guarantee SC.

We have already said that PLSC is a desirable property, because without it, executions become extremely non-intuitive. What about SC? Should we demand sequential consistency from every execution? It is easy to design a system that preserves SC and simultaneously guarantees high performance?

To answer this question, let us conduct a small experiment. Let us take a multicore processor such as a regular Intel or AMD machine and write the following piece of code using two threads: T_1 and T_2.

T_1	T_2
$x = 1$	$y = 1$
$t1 = y$	$t2 = x$

Here, x and y are global variables. Let us assume that all our global variables are initialized to 0. $t1$ and $t2$ are local variables stored in registers. The convention that we shall henceforth use is that all local variables that are stored in registers are of the form ti, where i is an integer.

Is the outcome $(t1, t2) = (0, 0)$ allowed? This behavior is not intuitive. We are in a better position to answer this question now. This is the same execution as that shown in Figure 9.10. This was proved to be not in SC.

Let us now run this piece of code on a real Intel or AMD machine where the two threads are assigned to two different cores. We shall observe that the outcome $(0, 0)$ is indeed observed! This is because almost no practical systems today are sequentially consistent. SC is a good theoretical concept and makes program executions appear intuitive. However, to support it we need to discard most architectural optimizations. For example, on Intel machines the write-to-read memory order (for dissimilar addresses) does not hold. This means that in Thread 1, the core can send the instruction $t1 = y$ to memory before sending $x = 1$. This will indeed happen because loads are immediately sent to memory, once the address is resolved. However, stores are sent at commit time and take effect later when they actually update the memory location.

Recall our discussion on start, end, and completion times. It is time to use these concepts now. In all likelihood, in an OOO core, the address of the succeeding load instruction $t1 = y$ will get resolved before the preceding instruction $(x = 1)$ commits. As soon as the address of the load to y is resolved, we will send the load instruction to memory. Hence, the completion time of the load to y will most likely be before the completion time of the store to x. This clearly violates program order but is a direct consequence of an OOO design, where we send loads to the memory system as soon as their address is resolved. This improves performance significantly because loads are often on the critical path. Unfortunately, this ensures that the completion times of these two instructions – an earlier write and a later read to a different address – are not in program order. This ensures that a parallel execution is not in SC, even though this does not create any issues with single-threaded executions. In fact, almost all architectural optimizations starting from write buffers to caches to complex NoCs reorder the execution of memory instructions in a program. For example, a write buffer allows later loads to go directly to the cache, but stops earlier stores from being written to the cache. Hence, in a multicore machine with OOO cores, SC often fails to hold. In general, whenever we advance the completion time of instructions

on the critical path such as load instructions to increase performance, we are essentially violating SC. We celebrated such optimizations when we were considering a single core running a single thread, sadly, they are singularly responsible for making executions non-SC in multicore systems.

We thus see that SC is the enemy of performance – it precludes the use of advanced architectural optimizations. Even though SC is a gold standard of intuition and correctness, it is difficult to enforce it in modern architectures. We need to disrespect it if we wish to use all our architectural tricks. Now, if SC is not respected, how do programs work? Why did we explain so much about SC, if SC is not meant to be enforced?

We need to read the next few sections on consistency and data races to precisely answer this question. Our basic philosophy is that even though SC is not respected at the architectural level, we somehow want to *trick* the high level programmer into believing that SC is indeed respected as long as she follows some rules.

So, where are we now? We have appreciated that SC is a great theoretical tool that unfortunately cannot be fully enforced in practice. However, it does give us a powerful method of reasoning about the correctness of parallel executions. We also looked at PLSC that is somewhat less restrictive and holds for some executions that are not in SC. We have till now not commented about the practicality of PLSC. Let us analyze it further before tackling the question of how to deal with architectures that do not enforce SC. Keep in mind that the final goal is to somehow trick the programmer into thinking that the underlying architecture only produces SC executions.

9.3.3 Exploring PLSC Further: Non-atomic Writes

Let us now complicate our situation even more. Consider a non-atomic write that does not appear to execute instantaneously from the point of view of the programmer. It does not have a single completion time (again from the point of view of the programmer). Rather it appears to execute at different points of time to different threads. We clearly do not have the issue of non-atomic writes in single-threaded systems.

However, in multithreaded programs that run on multiple cores it is possible that a write to x might be seen at different times by threads on other cores. One core might see it earlier than others. This will particularly be a problem when we are considering a distributed cache. The location for x might actually correspond to different physical locations on different caches within this large distributed cache. If we cannot keep all of them synchronized and propagate updates instantaneously, then it is possible that different threads running on different cores will perceive different completion times.

T_1	T_2	T_3
Wx1	while(x != 1){}	while (y != 1){}
	Wy1	Rx0

Figure 9.13: Non-atomic writes

Let us consider one such execution in Figure 9.13. Assume that the *while* loop terminates in the first iteration. The memory operations that the system observes are captured in the execution shown in Figure 9.14.

T_1	T_2	T_3
Wx1	Rx1	Ry1
	Wy1	Rx0

Figure 9.14: Non-atomic writes

Thread T_1 writes to x ($Wx1$). Thread T_2 observes this write ($Rx1$), and then writes to y ($Wy1$). Then, T_3 observes the write to y ($Ry1$), and finally reads the value of x. Assume that the *while* loop introduces a happens-before relationship between the condition that it reads and the code after the loop. This should be the case because we cannot terminate a loop unless its exit condition is true.

We have seen the following happens-before relationships till now: $Wx1 \rightarrow Rx1 \rightarrow Wy1 \rightarrow Ry1$.

Let us now consider $Rx0$. This takes effect after $Ry1$ because of the *while* loop. We thus have $Wx1 \rightarrow Rx1 \rightarrow Wy1 \rightarrow Ry1 \rightarrow Rx0$. If we remove the operations in the middle, we end up with $Wx1 \rightarrow Rx0$. This is not possible. The sequence of operations is not legal. Instead of $Rx0$, we should have had $Rx1$. However, since this is not the case, we can conclude that this execution is not in SC. Furthermore, the write to x is not atomic. It is visible to T_2, yet is not visible to T_3 at a later time – it is not associated with a single completion time.

Now the important question that we need to answer is, "Do we allow such an execution?" Whenever, we have non-atomic writes, we will be confronted with similar issues. There is no straight answer. However, before taking a decision, we need to keep in mind that many commercial systems such as IBM PowerPC and ARM v7 machines [Alglave, 2012] do not enforce write atomicity. In these architectures, writes to global variables are non-atomic, which basically means in this context that the write to x reaches Thread 3 late. In such systems, this execution will be correct. Even though this execution is not sequentially consistent, the architecture will allow this. Given that commercial systems exist that do not enforce write atomicity, we have no choice but to accept this execution.

How can this happen? This can happen if the variable x is stored at multiple locations in a distributed cache. Thread 2 receives the update to x ($x = 1$), yet Thread 3 does not receive it because the message to deliver the update gets stuck in the NoC. Thread 3 thus ends up reading the older value of x.

Let us use the gold standard, PLSC, that we had developed in the case of atomic writes to analyze this execution. It stated that if we consider the execution with respect to a single memory location, then it should be sequentially consistent. Let us see if this property holds. Let us break down the execution shown in Figure 9.14 into two sub-executions (see Figures 9.15 and 9.16), where the operations in each sub-execution access just a single location.

T_1	T_2	T_3
Wx1	Rx1	Rx0

Figure 9.15: Accesses with respect to x

T_1	T_2	T_3
	Wy1	Ry1

Figure 9.16: Accesses with respect to y

Even though the overall execution is not in SC, PLSC holds for each location. This is tempting us to declare PLSC a necessary condition for intuitive behavior in the case of non-atomic writes as well. Before we do so, let us consider one more execution in Figure 9.17.

T_1	T_2	T_3
Wx1	Rx1	Rx2
Wx2	Rx2	Rx1

Figure 9.17: Execution with non-atomic writes for a single location

Here, Threads 2 and 3 successively read from the same location, x. T_2 reads 1 and then 2. T_3 reads 2 first and then 1. This execution is clearly not in SC. Since writes are non-atomic, we can always argue that this execution should be allowed, because we have after all accepted the non-SC execution in Figure 9.13 that had non-atomic writes. We can say that the write $Wx1$ propagated to T_2 quickly, and then took a long time to reach T_3. We will have a reverse situation with $Wx2$. It arrived at T_3 early and arrived at T_2 late.

So, should this execution be allowed? In this case, the situation is slightly different. We have two successive writes to x: $Wx1$ and $Wx2$. They are made by the same thread, and two other threads see them in different orders. This means that x is perceived to have two different final states: 2 according

to T_2, and 1 according to T_3. How can the same variable have two different final states? Let's say that after a long time T_2 and T_3 read the value of x, and if there are no intervening writes to x, they will still read different values. This should not be allowed. This is breaking the notion of memory completely. x is no more associated with a single logical location. It is as if the two threads saw two different variables. Hence, let us conclude that this behavior should not be allowed. This is indeed the case. No commercial processor allows this.

Now why is this behavior different from the earlier example shown in Figure 9.13, where we decided to allow non-atomic writes? Let us try to answer this question using the PLSC constraint. The reader needs to convince herself that the execution shown in Figure 9.17 is not in SC and neither in PLSC. This means that even if an execution is not in SC because of non-atomic writes, some architectures still allow it because it satisfies PLSC. However, if the execution does not satisfy PLSC, it is not allowed.

What makes PLSC holier than SC? This has to do with the fact that ensuring SC is difficult at an architectural level mainly because it disallows many architectural optimizations. However, to enforce PLSC, we just need to ensure that an observer observing all the accesses to a memory location perceives a sequentially consistent order. This is simpler.

SC is like eating salad and doing exercise very day. This is ideal yet impractical! It is far better to somehow *trick* the body into believing that the person is actually doing this. PLSC is like popping a pill every day to keep cholesterol levels in check – this is far easier and doable !!!

Let us look at the PLSC vs SC issue further.

PLSC vs SC

What does sequential consistency entail? There are two aspects. We need to ensure that the operations issued by each thread take effect in program order, and secondly they appear to execute atomically or instantaneously (appear to have a single completion time). Sequential consistency is essentially program order + atomicity – this will allow us to arrange the operations of all the threads in a legal sequential order where intra-thread program order is respected. Note that reads to single memory words are always atomic because we can never have a *partial read* unlike a *partial write*, where some threads have received the updated value and some haven't. Hence, we normally say that SC = program order + write atomicity.

In OOO cores, ensuring program order for all memory accesses is difficult. However, it is far easier to ensure PLSC. We simply need to ensure that accesses to the same address are not reordered by the pipeline or the memory system. This is anyway the case as far as we have seen. We do not allow a load to go to the memory system if there is a prior store that writes to the same address. Loads always check the LSQ and write buffer to see if there are writes to the same address. Hence, our pipeline does not reorder memory accesses to the same address – later memory accesses do not overtake earlier memory accesses. The NoC may however do it. For example, two store operations to the same address issued by the same core may be reordered by the NoC – this needs to be stopped.

Now, let us look at write atomicity, which can be viewed differently. Given a write operation W and any other memory operation X, the order of completion times of W and X as perceived by all the threads should be the same. This means that all the threads should agree that W either completed before X, or after X, or the relative ordering does not matter – two threads should never make different conclusions.

Ensuring this for different memory locations is not easy because different parts of the memory system manage the accesses to different memory locations. For example, the different locations might be in different cache banks with their own controllers. However, ensuring this for a single location is much easier – in this case, we only care about the point of view an observer that is looking at accesses to a single memory location only. We can simply ensure that the accesses are serialized – appear to execute one after the other. This will ensure atomicity.

An astute reader may argue that if writes are atomic from the point of view of a single location, they should be atomic from all other points of view, even when we are considering accesses to multiple locations. However, as we have seen in our examples, this need not be the case. As we saw in Figure 9.14,

it is possible that when we consider multiple locations, writes to a single location might appear to be non-atomic; however, if we consider writes to any given location, they appear to be atomic. It is all about the point of view. For a single memory location, the observer sits on the memory location, and for multiple locations, the observer sits on the core. They see different things.

Let us now summarize. Given that PLSC is much easier to enforce than SC, PLSC has been accepted as a correctness criterion that all shared memory architectures need to provide. SC, on the other hand, is desirable but impractical. We nevertheless need to give the programmer an *illusory assurance* that the architecture somehow ensures sequential consistency. We will take up this problem after we wrap up the discussion on PLSC.

9.3.4 From PLSC to Coherence

It is time to wrap up the discussion on PLSC and provide a set of guidelines to hardware designers. PLSC at this point appears more mathematical than architectural.

PLSC has two properties: program order and write atomicity (all from the point of view of a single memory location). The program order aspect is complicated by the fact that the NoC may reorder messages sent by the same core (for the same address). However, most memory access protocols have easy fixes for this issue. They serialize such accesses. The other property is write atomicity; this is from the point of view of an observer who is only interested in accesses to the same memory address. We will discuss coherence protocols in subsequent sections that ensure a global order of writes (the atomicity aspect of PLSC) to the same address.

However, before doing so, let us understand the implications of PLSC on the behavior and broad design of hardware and software.

Software Implications of PLSC

For software writers, PLSC clearly specifies the executions that are allowed and the executions that are not. PLSC tells the software writer that regardless of the underlying implementation, each memory location is updated in program order, and if we do not consider other addresses, writes are atomic even on non-atomic machines. We can then write programs accordingly.

Hardware Implications of PLSC

The implications on hardware design are more important. It is easily seen that PLSC is both a necessary and sufficient correctness criterion for specifying the behavior of a cache or memory. Any external observer perceives a single physical location associated with a given memory address, regardless of the internal implementation.

PLSC thus provides a theoretical framework to create a distributed cache. If a given memory address has multiple associated memory locations in a distributed cache, then as long as we can maintain PLSC at the high level, the distributed cache will act as a unified cache. An observer on the memory location in a unified cache will see a legal sequential execution. If it sees the same for accesses to the same address in a distributed cache, then it will have no way of knowing whether the cache is unified or distributed. We can thus realize the gains of a distributed cache without changing our notions of correctness.

What does it mean for cache design? A distributed cache should look like a black box to the upper and lower levels. For example, if we hypothetically consider a design where the L1 cache is shared, the L2 cache is distributed, and the L3 cache is shared, then there should be no way for the programmer, or even the L1 and L3 caches to know that the L2 cache has a distributed design. This is a rather bad design from the point of view of performance, but it should still work correctly.

For this L2 cache, program order does not make a lot of sense; it does not perceive the existence of threads on the cores. Each of its constituent caches (known as *sister* caches) has a FIFO (first-in-first-out) queue that is used to accept memory requests. The requests in each FIFO queue are processed and

completed in FIFO order, which is the program order from the point of view of the L2 cache. Hence, its only job is to maintain write atomicity when we consider accesses to a single location.

What exactly does this mean from the standpoint of hardware that can see all the accesses to all locations? Let us go back to observers and points of view. Let us define a hypothetical external observer $\mathcal{O}$ that sees the accesses for only a specific memory location (as we have defined before). By PLSC, $\mathcal{O}$ sees a legal sequential execution. Next, let us attach an observer with each sister cache; with the i^{th} sister cache, let us attach observer $\mathcal{S}_i$. It creates a sequential execution for each address based on the times at which it receives messages on the NoC; unlike $\mathcal{O}$ it does not have a global view; it conveys the perspective of real hardware.

Let us analyze four sub-cases where we discuss the ordering between read and write operations to the same address that are issued by different sister caches.

Case I: Consider two read operations R_i and R_j to the same address that read the value produced by the same write (as per $\mathcal{O}$). Let us use the operator $\rightarrow$ to indicate the order of the accesses in a sequential execution. Does the order between R_i and R_j matter across the sequential executions recorded by the sister caches? The answer is NO. This is because they read the same value.

Case II: Consider two write operations W_i and W_j. If $\mathcal{O}$ records $W_i \rightarrow W_j$, then all the sister caches need to record the same order. Otherwise, read operations stand to get the wrong values, and the final value of the memory location will also be undefined. Since this does not happen in PLSC, the order of writes is *the same* across the sister caches.

Case III: Consider a read and a write operation: R_i and W_j. Assume that R_i returns the value written by W_j. As per PLSC, $\mathcal{O}$ will observe the order $W_j \rightarrow R_i$. If we have atomic writes, all the sister caches will also observe the same order. However, if we have non-atomic writes this need not be the case. It is possible that some sister caches may see the write *early* and thus not record $W_j \rightarrow R_i$. This means that for them, the completion time of the read will be before the completion time of the write. A write is said to complete when its value reaches all the sister caches and no subsequent read can read an older value. Hence, all the sister caches may not agree on such a write-to-read ordering.

Case IV: Consider a read and a write operation, R_i and W_j, where R_i reads its value from W_i, and W_j is ordered after W_i. As per PLSC, $\mathcal{O}$ will observe $R_i \rightarrow W_j$. Will the rest of the sister caches also record this order? Let us prove by contradiction. If a sister cache recorded $W_j \rightarrow R_i$, it would have been forced to return the value written by W_j or a newer value. This has not happened. Hence, all the sister caches must have recorded $R_i \rightarrow W_j$.

Let us summarize what we just learned. We learned that for a distributed cache that is built using FIFO queues, the way that we described, all that it needs to additionally do is ensure that for accesses by different threads to the same address, the read-to-write and write-to-write orders are *global* (all sister caches agree). This is captured by cases II and IV. Alternatively, this means that all the constituent sister caches view the *same order of writes to the same address*. The read-to-write ordering discussed in case IV is subsumed within this definition.

Axioms of Cache Coherence

Let us go a step forward and define the axioms of cache coherence as follows.

Write Serialization (WS) Axiom All the writes to the same address are seen in the same order by all the threads. This is derived from cases II and IV. This is also known as the **Order Axiom**.

Write Propagation (WP) Axiom A write is eventually seen by all the threads.

The write serialization axiom (WS axiom) captures the relevant part of PLSC in the context of a distributed cache. *Serialization* means a process where we observe a set of events, such as the writes to the same address, as a *sequence*. The write propagation axiom is new; it has not been discussed before. It is rather trivial in the sense that all it says is that a write never gets lost. It is ultimately visible to all the threads. We shall make use of these axioms to create cache coherence protocols in Section 9.4.

9.3.5 SC using Synchronization Instructions

Given that we have been able to manage the correctness of accesses to the same memory address with PLSC and cache coherence, we need to revisit the issue of sequential consistency that we have kept pending. It is sad that we have to make a choice between architectural optimizations that we worked so hard to design and SC. If we had SC, we could have executed the piece of code shown in Figure 9.18 seamlessly. This is one of the most common primitives in parallel programming: it allows a producer thread to communicate a value to a consumer thread.

T_1	T_2
value = 3;	while(status != 1){}
status = 1;	temp = value;

Figure 9.18: Communicating a value between threads

This piece of code will work perfectly in a sequentially consistent system. *temp* will always be set equal to *value*. Furthermore, thread T_2 will wait for thread $T1$ to set $status = 1$. Unfortunately, if SC does not hold, specifically if program order does not hold, then we may exit the *while* loop prematurely. We are not guaranteed to see $temp = value$. This is the primary mechanism that is used to communicate values between threads. The *while* loop is known as a *spin lock* or a *busy wait loop*.

Let us outline a software solution to ensure that this piece of code works correctly.

Synchronization Instructions

The most common synchronization instruction is the *fence* instruction, which is a special instruction that is present in almost all multiprocessor systems as of 2020. It artificially introduces an ordering between instructions. The orderings enforced by a fence instruction are as follows.

read $\rightarrow$ fence
write $\rightarrow$ fence
fence $\rightarrow$ read
fence $\rightarrow$ write
fence $\rightarrow$ fence

This means that all read and write instructions before the fence instruction (in program order) need to fully complete before the fence instruction completes. A read fully completes when it gets its value. Similarly, a write fully completes, when it reaches all the cores and the value cannot change henceforth. Handling reads is easy. We can consider it to be fully completed, when the value reaches the core. However, for a write, the only way to ensure that it has fully completed is to wait for an acknowledgement from the memory system. Secondly, later instructions (after the fence in program order) cannot start until the fence instruction has completed. Once a core decodes a fence instruction, it stops sending later instructions to memory. Once all the preceding instructions are deemed to have completed successfully, the core executes the fence instruction. A vanilla[1] fence instruction merely introduces an ordering. Once we commit a fence instruction, we can then start executing later read,

[1] ordinary or standard

write, and synchronization instructions. Note that if we just consider the execution of fence instructions, it is in SC.

Along with the basic fence instruction, there are other kinds of synchronization instructions that do other things as well such as atomically reading, modifying, and writing to one or more memory addresses. Nevertheless, almost all such variants still include the functionality of the fence operation that essentially ensures that all the instructions before it in program order fully complete before any instruction after it in program order completes – it enforces an ordering of completion times.

Use of Synchronization Instructions

A trivial solution is to make every instruction additionally behave like a fence if we would like an execution of a program to be in SC. ISAs such as x86 provide this facility by having a *lock* prefix, which makes some instructions additionally behave as a fence. However, such fences make the program very slow. Tudor et al. [David et al., 2013] estimate that executing a *fence* instruction takes 100-300 cycles, which is prohibitive. Hence, our aim is to insert the minimum number of fences in a program such that the behavior is predictable. Note that problems arise only while accessing shared variables. If we are accessing thread-local variables, then there is no need to add fences. We can use such insights to reduce the number of fences.

Let us explain with an example. Consider the piece of code that was originally shown in Figure 9.18. Let us now add fences such that all executions are in SC irrespective of the underlying memory model. The code is shown in Figure 9.19.

T_1	T_2
value = 3;	while(status != 1){}
fence;	fence;
status = 1;	temp = value;

Figure 9.19: Communicating a value between threads. The code has fences to make the code behave in a sequentially consistent fashion.

Irrespective of the underlying memory model, the execution will always be in SC. For example, if we do not respect the $write \rightarrow write$ order, it does not matter. Because of the fences, first the write to *value* will complete, and then the write to *status* will complete. When we exit the *while* loop we will be sure that *value* has been set correctly. We can happily set $temp = value$.

In general, figuring out the locations where we need to add fences, such that the number of fences is minimized and each execution is in SC, is a computationally intractable problem. It is often easy to find a sub-optimal solution, where we insert more fences than necessary.

Acquire and Release Synchronization Operations

There are a few other kinds of restricted fence operations that provide a subset of the functionality of the basic fence instruction.

Acquire instruction No instruction after the acquire instruction in program order can complete before it has completed. Note that an acquire instruction allows instructions before it to complete after it has completed.

Release instruction The release instruction can only complete if all the instructions before it have been fully completed. Note that the release instruction allows instructions after it to complete before it has completed.

Memory barriers Memory barriers are restricted fence operations, which disallow particular types of reorderings. For example, a write barrier such as *stbar* in the SPARC® ISA prevents $write \rightarrow write$ reordering. We have similar memory barriers for different kinds of instruction reorderings.

9.3.6 Theory of Memory Models

An important question that we need to ask ourselves is, "Should we give up on sequential consistency?" We did try to force SC in Section 9.3.5 with fence instructions. Even though this solution can guarantee SC, the overhead is prohibitive. A fence instruction takes 100-300 cycles, and we cannot add many such instructions.

We thus need to adopt a more nuanced approach and look at other methods to reason about parallel executions. So let us try to generalize the approach that we used to map a parallel execution to a legal sequential execution. Let us introduce an intermediate step. We first convert a parallel execution into some form of an intermediate representation, and then convert that to a sequential execution (not necessarily *legal*). An intermediate representation, which we shall refer to as an *execution witness*, can *map* a parallel execution to many sequential executions. We can use it to prove the correctness of a piece of parallel code, and also write parallel code that runs on a non-SC machine. This conceptual view is shown in Figure 9.20. An important point to note here is that the mapped sequential execution may or may not be legal. This is dependent on whether writes are atomic or not. Notwithstanding this limitation, the method of execution witnesses is a powerful technique that can be used to reason about memory models.

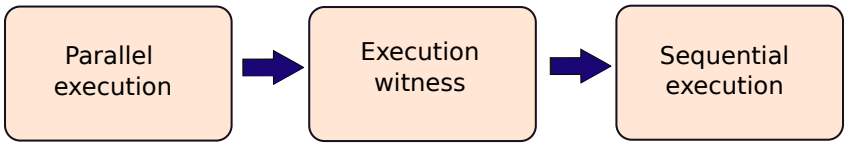

Figure 9.20: Parallel execution ⇒ Intermediate representation ⇒ Sequential execution

The rules for converting a parallel execution to an execution witness are dependent on the nature of program orders that are preserved in executions and whether stores are atomic or not as we saw with SC and PLSC. These rules pretty much govern the behavior of the memory system with respect to accesses to multiple memory locations. They comprise the *memory consistency model*, or in short, the *memory model*.

Let us summarize. The goal is to reach a sequential execution by any means. This is because as human beings, we find it far easier to reason about sequential executions. A sequential execution has the notion of an earlier write and a later read. It essentially captures the view of an omniscient observer that is aware of the completion times of all the operations. Of course, given a parallel execution we will never be able to guess the completion times with certainty. Nevertheless, a mapped sequential execution represents one possible order of completion times that is consistent with the parallel execution and the memory model. We might have many such mapped sequential executions that all read and write the same values. The important point is that it should be possible to find at least a single one such that we can argue that there is some order of completion times that can explain the parallel execution. If we can map a parallel execution to a sequential execution as per a memory model, the parallel execution is said to be *feasible* under that memory model. Furthermore, if a piece of parallel code satisfies a certain property such as a given variable should always be set to 1, then all the parallel executions will satisfy the same property, and all mapped sequential executions will do the same too. It does help if the mapped sequential execution is legal – it makes the execution seem more intuitive. Even if it is not, the sequential execution still provides important insights and can be used to verify if a machine follows a given memory model or not.

Memory Consistency Models

Definition 71
The rules governing the behavior of cores and the memory system while accessing multiple memory locations is known as the memory consistency model *or the* memory model. *It defines the set of valid outcomes for any program: sequential or parallel.*

Given the importance of the memory model in a computer architect's life, it is essential that she understands the basics of a memory model really well. There are two ways in which we can study memory models: from a hardware designer's point of view and from a programmer's point of view. The earlier approach, which is from the hardware designer's point of view was far more common till 10 years ago (as of 2020). In this case, researchers focused on the way that we implement different memory operations in the memory system, and what exactly is allowed, and what is not. The problem with this line of approaches is that it does not convey the big picture to students, and it does not arm them with theoretical tools that they can use to analyze programs, executions, and hardware systems.

Hence, the other approach, which is to just look at program behavior from the programmer's point of view is far more prevalent these days particularly with programming language researchers and members of the verification community. We shall adopt this line in our book and present a theoretical framework to understand different memory models. In specific, we shall use the framework proposed by Alglave et al. [Alglave, 2012]. Her model covers all existing memory models as of 2020, and is fairly generic enough to be extended for future models as well.

An Execution Witness

Let us introduce the basic terminology proposed by Alglave [Alglave, 2012]. Given a parallel program, let us only consider the different types of memory operations: read, write, and *synch* (synchronization) operations.

The rest of the operations need not be considered. Since the cores that execute the parallel threads do not run in lockstep, they can get delayed for indefinite periods, and thus we cannot guarantee the relative timing of the operations. As a result in different runs, we may have different outcomes. The space of all possible outcomes is determined by the memory model. A single run is a parallel execution or just an *execution* (formally defined in Section 9.3.1). For example, the code in Figure 9.21(a) can have two different executions (on an SC machine): see Figures 9.21(b) and 9.21(c).

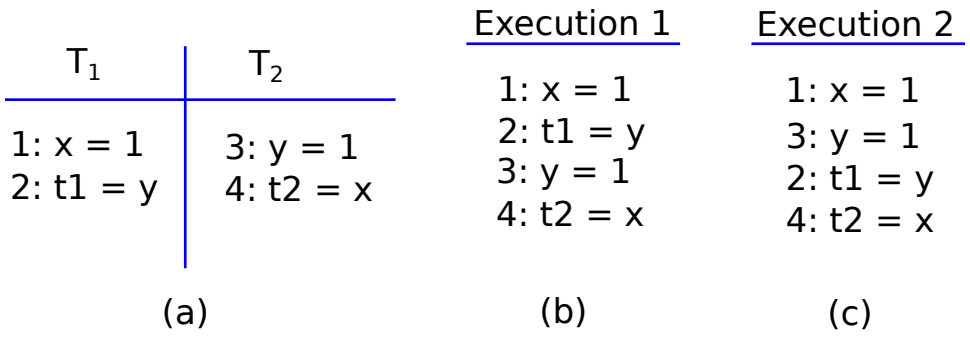

Figure 9.21: Two different SC executions of a parallel program

The space of all possible executions for a given program in a given system is known as the space of *valid executions*. For each execution, we can create a graph called the *execution witness*. Recall from

Section 2.3.2 (Definition 11) that a graph is a data structure with nodes and edges, where nodes or vertices are connected with edges (similar to a network of roads where cities are the nodes and the roads are the edges).

In the case of an execution, the nodes are the memory accesses (read, write, or synch), and the edges are the relationships between the nodes, which we shall define shortly. We can use this information to create the execution witness for the execution, which can be further analyzed to understand the features of the execution, and its interaction with the memory model. An execution witness is a nice graphical tool that can be used to understand if an execution is valid or not, and the limitations that a memory model imposes on the architecture (and vice versa). We shall primarily use four kinds of edge labels in the execution witness: rf, po, ws, and fr. Note that we can have multiple execution witnesses for an execution, in that case, we only consider that witness (if there is one) that is allowed as per the memory model. In most of our examples, we will only have a single execution witness for a given execution, and thus this problem will not arise.

For defining these edges we will show examples of parallel code and their associated execution witnesses. The conventions that we shall use are as follows.

1. All global variables start with alphabets (other than 't'), and are initialized to 0.
2. All thread-local variables that are restricted to a given thread, start with 't'. They are typically stored in registers.

Nature of Edges: Global and Local

Before introducing the different kinds of edges, let us define a crucial property of the edges. We add an edge between two nodes (memory accesses) in the execution witness if we believe that a memory access (generalized as an *event*) happened before the other. When we want to say that event A happened before event B, we write $A \overset{hb}{\rightarrow} B$, and add an edge from event A to event B. Consider an example in Figure 9.22 for a sequentially consistent system, where we show a parallel program with two statements, an execution that shows that operation $Wx1$ executed or completed before (happened before) operation $Wy1$, and then an execution witness. The execution witness has two nodes, $Wx1$ and $Wy1$, and an edge between them that indicates the happens-before relationship between them.

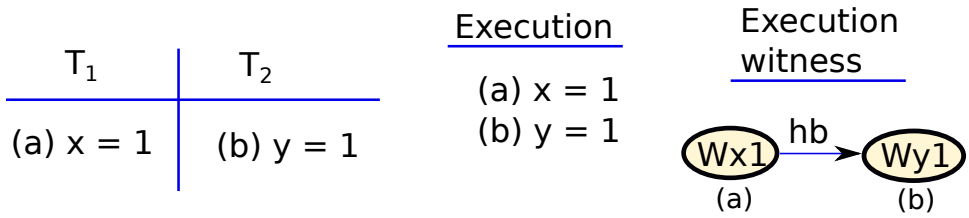

Figure 9.22: Example of an execution witness (assume a sequentially consistent system)

A happens-before relationship can be of two types: local or global. If a thread believes that event A happened before event B, then we can write $A \overset{lhb}{\rightarrow} B$ (A happened before B). From the point of view of that thread (T_1), this relationship holds. We can alternatively write, $A \overset{hb}{\rightarrow} B \in lhb_{T_1}$. In this case it is possible that another thread T_2 might have a different view and might observe $B \overset{lhb}{\rightarrow} A$. This is the local view of T_2. To summarize, a local view does not necessarily hold across threads. There is no global consensus.

When we say that the relation $A \overset{hb}{\rightarrow} B$ is global, it means that all the threads agree that A happened before B. There is no disagreement between two threads. In this case, we can write $A \overset{ghb}{\rightarrow} B$ or $A \overset{hb}{\rightarrow} B \in ghb$.

Further, note that the $\overset{hb}{\rightarrow}$ relationship does not indicate if the relationship is local or global – this has to be interpreted from the context. In this book, we shall use the symbol $\overset{hb}{\rightarrow}$ if its scope (local or global) can be interpreted from the context or if it does not matter.

Definition 72

- *A is said to globally happen before B, if all the threads agree with the fact A has happened before B.*
- *Every thread has a view of the events. It is possible that a given thread T_1 may feel that $A \overset{hb}{\rightarrow} B$, and another thread T_2 may feel that $B \overset{hb}{\rightarrow} A$. In this case, the relationship $A \overset{hb}{\rightarrow} B$ is local to T_1 and is not global. We thus write $A \overset{lhb}{\rightarrow} B$.*
- *If the happens-before relationship holds globally (across all threads), then we write $A \overset{ghb}{\rightarrow} B$.*

All variants of the $\overset{hb}{\rightarrow}$ relationship are transitive relationships, which means that $A \overset{hb}{\rightarrow} B$ and $B \overset{hb}{\rightarrow} C \Rightarrow A \overset{hb}{\rightarrow} C$. A set of happens-before edges between events recorded by the same observer cannot have a cycle. This means that we cannot have a set of relationships as follows: $A \overset{hb}{\rightarrow} B$, $B \overset{hb}{\rightarrow} C$, and $C \overset{hb}{\rightarrow} A$. This would automatically imply that $A \overset{hb}{\rightarrow} A$, which is not possible (using the transitivity property). The fact that a graph with happens-before edges cannot have a cycle will be used extensively to understand multiprocessor systems. For similar reasons, any graph with just $\overset{ghb}{\rightarrow}$ edges cannot have a cycle: it would imply that an event happened before itself, which is not possible.

Relations and Union/Intersection Operations

In Computer Science a relation is a set of tuples, which in this case is defined as a pair of events. Consider a set of memory operations. Let us refer to each memory operation as an event. Let the events be A, B, C, D, and E. Assume that there is a happens-before relationship between them, and this fact can be represented as: $A \overset{hb}{\rightarrow} B \overset{hb}{\rightarrow} C \overset{hb}{\rightarrow} D$. We can say that the $\overset{hb}{\rightarrow}$ operation defines a relation hb, which is simply a set of tuples of events of the form $\langle X, Y \rangle$, where $X \overset{hb}{\rightarrow} Y$. For this particular example, the relation $\overset{hb}{\rightarrow}$ is defined as a set of the following tuples (note the happens-before relationship in each pair).

$$\langle A, B \rangle, \langle A, C \rangle, \langle A, D \rangle, \langle B, C \rangle, \langle B, D \rangle, \langle C, D \rangle$$

Specifying a relation such as $\overset{hb}{\rightarrow}$ using a list of tuples of events is another way of defining the relationship. This is however very cumbersome; nevertheless, it helps in understanding it from a theoretical perspective. We can similarly define another relation xy with the following tuples: $\langle B, C \rangle$, and $\langle B, E \rangle$.

Now, we can define a union of relations, which is similar to a union of sets, where the result contains all the tuples that are contained in at least one of the relations. The symbol for union is $\cup$.

$$hb \cup xy = \langle A, B \rangle, \langle A, C \rangle, \langle A, D \rangle, \langle B, C \rangle, \langle B, D \rangle, \langle C, D \rangle, \langle B, E \rangle$$

We can similarly define intersection, where the intersection of two relations consists of only those tuples that are present in both the relations. The symbol for intersection is $\cap$.

$$hb \cap xy = \langle B, C \rangle$$

Let us now discuss the different kinds of edges we add in an execution witness. All of these are happens-before edges.

Definition 73
A relation (R) between two sets A and B is defined as a set of pairs (2-tuples) of elements, where the first element is from set A and the second element is from set B. A and B can also refer to the same set. Consider an example. Let us have a relation IsTallerThan defined over the set of students in a university. Each student is represented by her name (assume it is unique). Then, if we write $IsTallerThan(Harry, Sofia)$, it means that Harry is taller than Sofia.

We can define all kinds of set operations between relations such as union and intersection. These are similar to union and intersection operations on regular sets.

We can also say that a relation R_1 is a subset of relation R_2, if all the tuples that belong to R_1, also belong to R_2, but not necessarily the other way. We write $R_1 \subset R_2$. If there is a possibility that R_1 and R_2 might be the same, then we write $R_1 \subseteq R_2$.

Consider an example. Let us define a relation $IsTallerBy2ft$, which contains all pairs of people where the first person is taller than the second person by at least 2 feet. It is clear that $IsTallerBy2ft \subseteq IsTallerThan$.

Program Order Edge: *po*

The program order (*po*) edge is a happens-before edge between memory operations issued by the same thread. They need not have the same address. In an SC execution, where operations complete in program order, we add *po* edges between consecutive operations issued by the same thread. In other memory models, we add edges depending upon the kind of orderings that are allowed. There are six kinds of *po* edges.

1. po_{RW} (read to write): This edge is between two memory operations, where the first operation is a load and the second is a store.
2. po_{WW} (write to write): *po* edge between two store operations.
3. po_{WR} (write to read): The first operation is a store, and the second operation is a load.
4. po_{RR} (read to read): *po* edge between two load operations.
5. po_{IS} (read/write to synch operation): Edge between memory operations and a subsequent synch operation.
6. po_{SI} (synch operation to read/write): Edge between a synch operation, and subsequent memory operations.

We are reiterating the fact that these dependences can be between operations with different memory addresses. The only thing that matters is their relative order within the thread. We shall see in subsequent sections that different memory models give different degrees of importance to different types of program order edges. For example, in the x86 memory model, po_{WW} edges are global, whereas they are not global in the PowerPC and ARM memory models. Depending upon the subset of *po* edges that are global, we can afford different kinds of optimizations in the pipeline and the memory system. We shall use the symbol $\stackrel{po}{\rightarrow}$ to refer to *po* edges in an execution.

However, if there are synch operations in the program, there is an edge between the synch operations and other regular read/write operations. These edges are global in nature. This means that there needs to be a consensus among all the threads that the synch operation completes only after all the operations

before it in program order, and furthermore, all the operations after the synch operation complete after the synch operation completes.

Figure 9.23 shows an example of program order edges in an execution witness. We show the execution of a single-threaded program on an SC machine, which preserves the ordering between the operations. The rounded and shaded box with the text $t1 = 1$ shows the outcome of the execution, which is that $t1$'s value is 1.

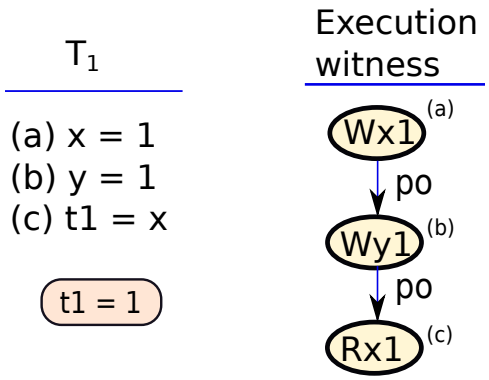

Figure 9.23: Example of program order (po) edges (execution of a single-threaded program on an SC machine)

Read-from Edge: rf

The rf (read from) edge captures a data dependence for reads/writes to the same address either in the same thread or across threads. If we have a read operation R, and a write operation W, where the read operation reads the value written by the write operation, then we have a dependence between the read and the write. It is a happens-before relationship because the write W needs to complete first, before the read operation can read its value. Since the read operation has read its value, we can automatically infer $W \xrightarrow{hb} R$. This is called a read-from relationship or an rf relationship and can be captured with a new type of edge in the execution witness. Let us refer to this as the rf edge, and denote it by $\xrightarrow{rf}$. We thus have $W \xrightarrow{rf} R$.

Figure 9.24 shows an example of a dependence where a write operation sends data to a read operation in a different thread. It is not necessary that the read operation belong to a separate thread, it can also belong to the same thread. In both cases, we shall have a read-after-write or an rf dependence, which is a happens-before relationship.

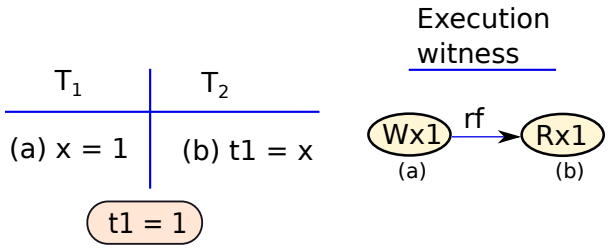

Figure 9.24: An execution witness with the rf edge

Let us divide the relation rf into two sub-relations: rfi and rfe. The rfi relation (read from

internal) is a write to read dependence in the same thread. In other words, the read and the write are operations issued by the same thread. The rfe relation (read from external) also represents a write to read dependence; however, in this case the read and write operations are issued by different threads. We have $rf = rfe \cup rfi$, where $\cup$ stands for set union.

The two rf relations, rfi and rfe, need not be global. This depends on the memory model. For example, if a write is non-atomic, it will be visible to some threads earlier than it is visible to other threads. This would automatically mean that the rfe relationship does not hold globally because all the threads will not agree on the order of operations. We shall explore the intricacies of such issues along with their architectural implications in later sections. Finally, note that in many places we shall use the generic term rf which can stand for either rfe or rfi or both. The nature of the usage will be clear from the context.

Write Serialization Edge: ws

Let us now look at the edges that we need to add because of PLSC and cache coherence. As we had discussed in Section 9.3.4, there is a global order of writes to the same address – all the threads agree with this order.

Let us thus add an edge between two write accesses to the same address and call it a write serialization edge, or a ws edge, denoted as $\overset{ws}{\rightarrow}$. It is a global happens-before relationship (follows from the write serialization axiom of coherence).

Figure 9.25 shows an example of the ws relationship, where we have two writes to the same variable x. Even though the writes are performed by different threads, there is still an ordering between them to satisfy PLSC, and thus we add a $\overset{ws}{\rightarrow}$ edge between them.

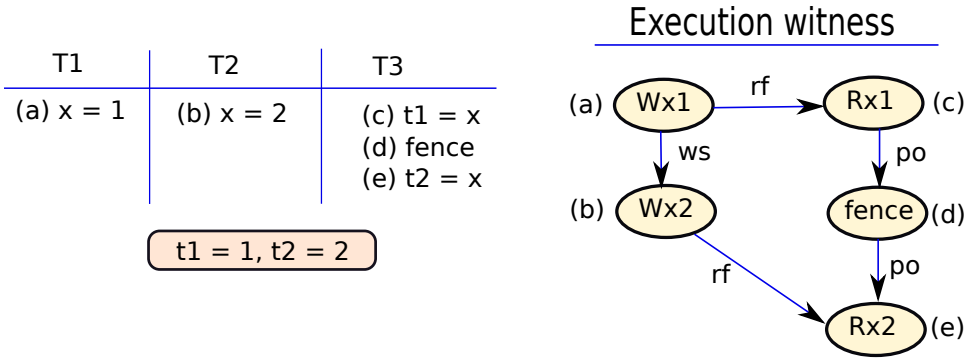

Figure 9.25: An execution witness with the ws edge

From-read Edge: fr

Let us now discuss another kind of edge that arises as a natural consequence of PLSC and the axioms of coherence (this was discussed in Section 9.3.4). We shall refer to it as the $read \rightarrow write$ or simply the fr (from-read) edge.

Consider the piece of code shown in Figure 9.26 and its associated execution witness, where we have two writes to a variable, and one read. In this case, we have a ws dependence between operations $Wx1$ and $Wx2$ because they write to the same variable x and the write operation $Wx2$ is the later write.

However, in this case, we have an intervening read operation $Rx1$ that reads the value of the first write operation $Wx1$. There is an rf edge between the operations $Wx1$ and $Rx1$. However, between $Rx1$ and $Wx2$, we have a dependence. $Wx2$ needs to happen after $Rx1$, otherwise we would read the value of x to be 2, which is not the case. Since there is an order between $Wx1$ and $Wx2$ due to PLSC,

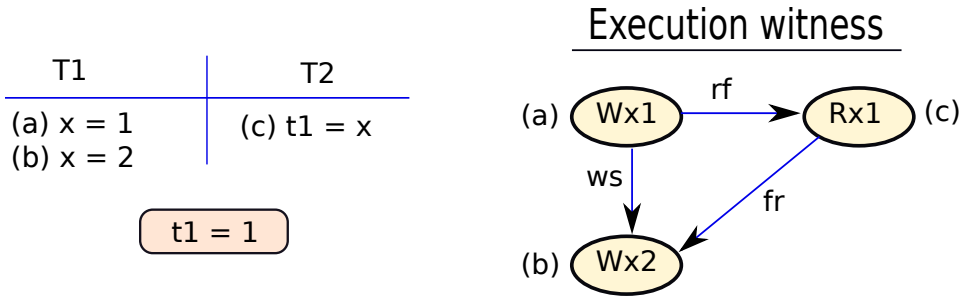

Figure 9.26: An execution witness with the fr edge

by implication, we have an order between $Rx1$ and $Wx2$ as well. Let us create an edge to represent such a read-to-write relationship, and name it the fr edge (represented as $\xrightarrow{fr}$). Akin to the ws relationship, the fr relationship is also global. Otherwise, PLSC will not hold (proved in Section 9.3.4).

Synchronization Edge: *so*

We assume that all synchronization operations are globally ordered with respect to each other. If we just consider all the synch operations, the execution is sequentially consistent. Recall that along with fence operations, we can have many more synch operations that additionally read or write to memory addresses (synch variables). For synch operations, we assume that rf and po are global. Furthermore, because of PLSC, ws and fr are also global.

Whenever we show an execution witness, we shall indicate the regular variables and the synchronization variables (exclusively accessed by synch operations). ws, rf, and fr edges between accesses to the synchronization variables will always be added. In some cases, it will be necessary to highlight the fact that we are adding an edge between accesses to a synchronization variable. In this case, we will additionally annotate the edge with the symbol, so (or $\xrightarrow{so}$).

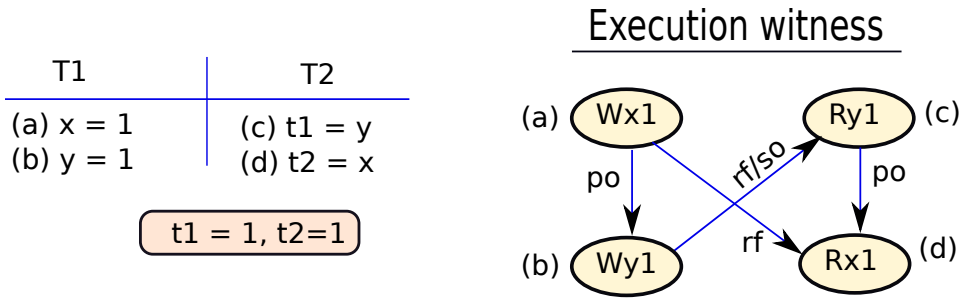

Figure 9.27: An execution witness with the so edge

Figure 9.27 shows an example. Here, x is a regular variable and y is a synchronization variable. We have an rf edge from $Wy1$ to $Ry1$. We additionally annotate the edge with the so symbol.

Conditions for Correctness

We have four types of relationships: po, ws, rf, and fr. ws and fr originate from PLSC; they are global relationships. The po and rf relationships might have some global and some local components (sub-relationships). Let $gpo \subseteq po$ be the subset of po that is global and similarly let $grf \subseteq rf$ be the

subset of rf that is global. A relation R_1 is a subset of relation R_2, if we can say with certainty that a tuple (pair of events) that is a part of R_1 is also a part of R_2. The reverse may not be true (see Definition 73). For example, it is possible that the po_{SI}, po_{IS}, and po_{WW} relations are global, yet the po_{WR}, po_{RW}, and po_{RR} relations are local. In this case, $gpo = po_{SI} \cup po_{IS} \cup po_{WW}$. Similarly, we may have a model where the rfe relation is global, but the rfi relation is local. In this case, $grf = rfe$.

Let us now define a correctness condition for an execution witness. If we have an execution witness with the edges from gpo, grf, fr, and ws, there should be no cycle. This is because we cannot have a cycle of global happens-before edges. A cycle of the form $A \overset{ghb}{\rightarrow} B \overset{ghb}{\rightarrow} C \overset{ghb}{\rightarrow} A$ implies that $A \overset{ghb}{\rightarrow} A$, which is not possible.

Alternatively, the overall global happens-before relation ghb can be written as

$$ghb = (gpo \cup grf \cup fr \cup ws) \tag{9.6}$$

ghb needs to be acyclic for every valid execution witness: it precisely characterizes the memory model.

Sequential Consistency

Sequential consistency is rather special when it comes to the four relations that we have defined. Since program order needs to be respected, $gpo = po$. Similarly, we have atomic writes; hence, $grf = rf$. The other two relations, fr and ws, need to hold anyway because they hold for all systems that respect PLSC.

Hence, we can write

$$SC = (po \cup rf \cup fr \cup ws) \tag{9.7}$$

Implications of an Acyclic Execution Witness

Let us recount our journey. We realized that we will not be able to create an equivalent legal sequential execution for every parallel execution on a non-SC machine. Hence, from a parallel execution we created an execution witness. Note that the execution witness need not be unique. However, the execution witnesses that we create have to be acyclic. This is because we cannot have a cycle of happens-before edges. This is a necessary property of a valid execution witness.

However, we have still not looked at the elephant in the room. If we are able to create an acyclic execution witness, where do we go from there? What does it give us? This is where we need to use a very important result from computer science: Theorem 9.3.6.1. We state the theorem without its proof.

Theorem 9.3.6.1 *In any acyclic graph, we can lay the nodes one after the other in a sequence such that if there is a path from node A to node B in the graph, then A appears before B in the sequence. The topological sort algorithm can be used to create such a sequence.*

There we go! We can create a sequential execution out of an execution witness. It will respect all the ordering relationships of the memory model and the execution witness. If there is a path of happens-before edges from operation A to operation B, then A will appear before B in the sequence. Since this sequence captures global orders, it may not be *legal*, particularly if the rf relation is not global. Nevertheless, it is a sequential order of operations, which is what we wanted to create – a sequential order presents a possible order of completion times of the instructions. It proves that an execution is *feasible* under a certain memory model. In fact, one of the classic ways of showing that a given memory model will not lead to a certain outcome of an execution is by showing that all execution witnesses will have a cycle – we will not be able to construct a sequential execution from them.

What can we use this sequential execution for? Given a piece of parallel code, we can find all the sequential executions for a memory model, either manually or using an automated tool. We can then use them to reason about the set of valid outcomes and check if a given property holds across all possible executions.

Let us reconsider this equation again: $SC = (po \cup rf \cup fr \cup ws)$. The po relation essentially means that in the mapped sequential execution, all the instructions of a thread appear in program order. Furthermore, $rf \cup fr \cup ws$ ensures that this sequential execution will be *legal* – there will be a global order of writes that appear to be atomic, and every read will get the value of the latest write.

9.3.7 Safety Conditions for Accesses to a Single Location

Let us now create a similar theoretical framework for the PLSC criterion based on the concepts that we have learned.

Correctness Conditions for Single-Threaded Programs

We have till now discussed the execution of multithreaded programs. We need to ensure that single-threaded programs still work properly when running on a multicore system. The execution of single-threaded programs needs to be independent of the underlying system and its memory model. In fact, a programmer should not be able to know if the program is running on a uniprocessor or multiprocessor system. To ensure that this abstraction is met, we need to add some more edges and constraints to ensure that this genuinely happens. Let us consider a single-threaded program as shown in Figure 9.28.

```
1 x = 1;
2 x = 2;
3 y = 3;
4 z = x + y;
5 x = 4;
```

Figure 9.28: Example program to show memory access constraints in uniprocessors

In this case, we are setting the value of z, after reading x and y (see Line 4). The variable x is initialized to 1 and then set to 2. In a single-threaded execution, from the point of view of the programmer, our requirements are as follows for computing z.

1. In the statement $z = x + y$, we read the value of x to be 2.
2. In the statement $z = x + y$, we read the value of y to be 3.

We can say that in every statement, we need to read the latest value of each operand (as per program order). As far as we are concerned there is no other requirement. As long as this condition holds for all variable and array accesses, we are fine. After all, the only basic actions that we perform are memory read, memory write, branches, and ALU operations. Branches and ALU operations are independent of the memory model. As long as we read the latest value that was written, the execution is correct.

Let us look at the happens-before edges that we need to have to ensure that this happens. Consider the variable x. We set it to 1, then to 2, we read its value to compute z, and then we finally set it to 4. These statements need to execute in program order, at least from the point of view of the current thread. If the order gets mixed up, then the final execution will be incorrect. Note that we can reorder the write to y (Line 3) with respect to the writes to x as long as it is done before y is read in Line 4. The execution will still be correct.

Let us now try to derive a pattern from this observation. For an execution to be correct on a uniprocessor, every read has to get the correct value, which is the value of the latest write to the same address. This means that we cannot reorder accesses to the same variable where at least one of them is a write. For the sake of simplicity, let us constrain all the accesses to the same variable to appear to an external observer as if they are happening in program order. Let us refer to this as the *uniprocessor access constraint where accesses to the same variable in a thread are not reordered and furthermore they take effect in program order.* This is the same as PLSC in the context of a single thread.

We can always reorder accesses to different variables such as reordering accesses to x and y in Figure 9.28.

Now, let us take this single-threaded program and run it on a multiprocessor. As long as the uniprocessor access constraint holds, the program will yield the same output irrespective of the memory model. All the reads will get the values of the latest write, and thus the execution will be the same. Let us thus create a new edge called a *up* edge (uniprocessor edge) that we can add between two operations that belong to the same thread and access the same address. We shall assume that the *up* edge is global when we are considering the point of view of an observer sitting on the memory location. She only observes the accesses to that specific memory location. We shall represent this edge with the symbol $\overset{up}{\rightarrow}$.

Access Graphs

An access graph is in principle similar to an execution witness. It can be used to deduce the correctness of programs running in multithreaded environments. Like the execution witness it also needs to be acyclic. The key differences in an access graph are that it contains accesses for only a single location, and the edges that we consider are up, ws, fr, and rf. The up edge enforces the uniprocessor access constraint for each thread. The rest of the edges show the constraints governing the communication of values across threads or the data flow. An observer sitting on a memory location will see all of these edges.

An example access graph is shown in Figure 9.29 for the code shown in Figure 9.28. Note the positions of the $\overset{up}{\rightarrow}$ edges that are added to ensure the uniprocessor access constraints.

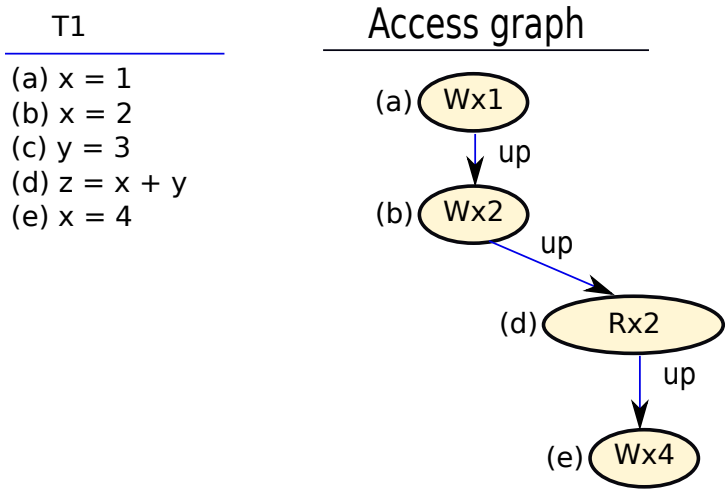

Figure 9.29: An access graph for the code shown in Figure 9.28

PLSC

Consider the four types of edges in the access graph: the uniprocessor access constraint up, and the three edges (ws, fr, and rf) that reflect the data flow between threads. They represent the behavior of the program from the point of view of a single memory location. Irrespective of the way these accesses interact with accesses to other locations, we would like all these four orderings to hold from the point of view of the single location. This is because they are required to ensure PLSC. In this case, the up relation represents the program order in executions that access a single location, the same way the po relation represented the program order in general programs. We have proven that $SC = po \cup rf \cup fr \cup ws$. We can define something similar for PLSC. The proof is on similar lines. For SC, we considered multi-variable executions. Now for PLSC, we shall consider executions that access a single variable. After replacing po with up we get

$$PLSC = up \cup rf \cup fr \cup ws \tag{9.8}$$

Example 11

Consider the code in Figure 9.30(a). Here, the two threads see the two updates to x in different orders. This is not allowed as per PLSC and coherence. To disallow an execution in our framework we need to find a cycle in the access graph.

Consider the access graph in Figure 9.30(b). We have a cycle between the nodes (c), (e), and (f). Since we cannot have a cycle with happens-before edges, this execution is not allowed. This execution is not in PLSC.

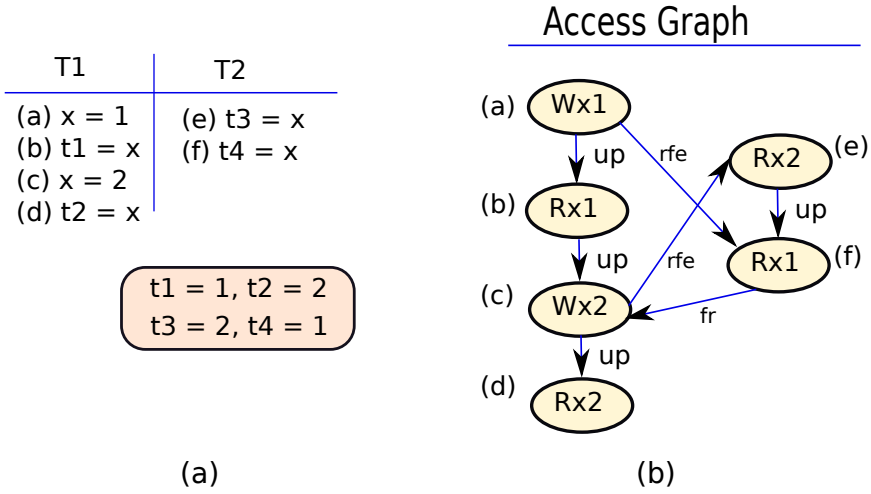

Figure 9.30: An access graph that shows an execution that does not satisfy PLSC

9.3.8 Safety Conditions for Data and Control Dependences

Even in models that do not respect program order, there is some causal dependence. A causal dependence is between a producer and a consumer (cause-effect relationship). It is not possible for a consumer to use a value before it has been produced. Till now, we have not captured such dependences. This can lead to extremely non-intuitive behavior.

Consider the execution in Figure 9.31(a) and the execution witness in Figure 9.31(b). In this case, $Rx1$, the *if* statement, and $Ry0$ are causally related: there is a cause-effect relationship. $Rx1$ is the producer of $t1$, the *if* statement is the consumer, and its consumer is $Ry0$. We can think of the *if*

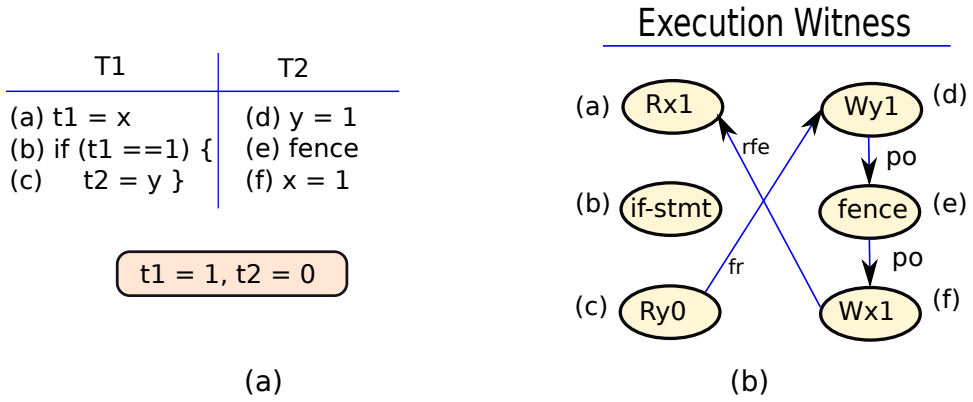

Figure 9.31: Non-intuitive execution when there is a data dependence

statement creating the permission for $Ry0$ to execute. Since our memory model does not respect read-after-read dependences, and does not treat *if* statements in a special manner, we have not added edges between $Rx1$, the *if* statement, and $Ry0$. However, in our memory model, we respect program orders between normal instructions and fences, and the rfe relation is global. Hence, we have added the corresponding edges. This execution witness does not have cycles and thus satisfies the memory model. However, it is not intuitively correct.

Instruction $Rx1$ produces the value of $t1$, which determines the direction of the *if* statement. Since in our execution $t1 = 1$, $Ry0$ executes. $Ry0$ reads $y = 0$ and thus there is an fr edge between it and $Wy1$ (in thread 2). The three instructions in thread 2 have to be executed in program order because the second instruction is a fence. After collating the dependences, we can conclude that $Wx1$ should complete after $Rx1$ because of the causal dependences, fr edge, and the fence. However, this is not what is happening. $Wx1$ produces the value for $Rx1$. Intuitively, we have a cycle even though we cannot see it in Figure 9.31(b). It appears that we have read $Rx1$ much before we should have actually read it. This is known as a *thin air read*. This can indeed happen in modern systems that use value prediction. If we had predicted the value of x to be 1, we would have later on found the prediction to be correct because of $Wx1$ and the execution would have been deemed to be absolutely fine!

Definition 74
A thin air read is defined as a read, where we read a value without seeing its preceding write.

Let us thus introduce a new edge called a dependence edge (*dep*) (symbol: $\overset{dep}{\rightarrow}$) that represents both data and control dependences. We add this edge between a read and a subsequent instruction that uses its value in the same thread or between a conditional statement and its body.

Let us thus create a new kind of graph to model causality. Let us call this a *causal graph* that only contains edges to model producer-consumer relationships. We have three kinds of edges in such a graph: rf edges, *gpo* edges and *dep* edges. *gpo* edges are program order edges that hold globally ($gpo \subseteq po$). Akin to the execution witness and access graph, the causal graph should also be acyclic.

Now, if we add these edges to the execution in Figure 9.31, we have the execution shown in Figure 9.32. We have added *dep* edges between $Rx1$, the *if* statement, and $Ry0$. Here, there is a cycle and thus the execution is not valid.

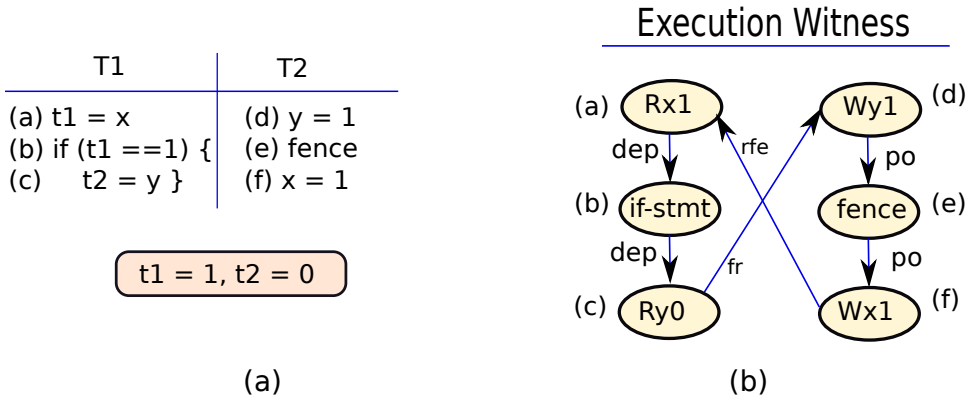

Figure 9.32: Example of an execution with the $\overset{dep}{\rightarrow}$ edge

Let us thus define a new condition that precludes thin air reads.

$$NoDepCycle = acyclic(rf \cup gpo \cup dep) \tag{9.9}$$

9.3.9 Correctness of Executions

Now we are in a position to answer the question, "When is an execution correct?" This is in general a tricky question because this depends on how exactly different systems, programming languages, and runtime environments exactly define correctness. Let us however take the liberty of providing a definition that enjoys a broad consensus in the technical community and is as per our discussion.

An execution is correct if all the following conditions in Table 9.2 hold.

Condition	**Test**
Satisfies the memory model	The execution witness is acyclic
PLSC holds for all locations	The access graphs are acyclic
NoDepCycle holds	The causal graph is acyclic

Table 9.2: Correctness conditions for an execution

9.4 Cache Coherence

Recall our discussion in Section 9.3.4 where we motivated the need for PLSC and showed how the axioms of coherence arise as a natural corollary of PLSC. In this section, we need to design a practical cache coherence protocol that ensures that the two cache coherence axioms hold: there is a global order of writes (write serialization), and a write is never lost (write propagation).

9.4.1 Write-Update Protocol using a Bus

Let us consider a simple system. Consider Figure 9.33, where the set of caches are connected using a bus. Recall that a bus is a set of copper wires that can support only one sender at any point of time; however, we can have any number of receivers. Such buses naturally support broadcast based traffic. It is possible for any cache to snoop on the bus – read all the writes that are made on the bus even if the

writes are not meant for it. Such a bus is known as a snoopy bus. Cache coherence protocols that use snoopy buses are known as *snoopy protocols*.

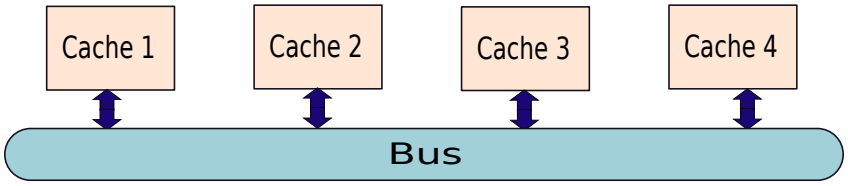

Figure 9.33: A broadcast bus

We shall discuss two kinds of snoopy protocols: write-update and write-invalidate. Let us describe the simpler protocol write-update in this section. The key idea is that every write is broadcast on the bus such that the rest of the caches can snoop it and take some action.

Each constituent sister cache, $C_1 \dots C_n$, in the distributed cache is a complete cache in itself. It can store any block. It can supply a copy of the block if it receives a request. However, to ensure that the set of caches follow coherence, we need to observe some rules. Before framing the rules, let us understand the constraints. Since $C_1 \dots C_n$ are mostly independent caches, they can have different copies of the same block. In this case, ensuring write serialization (WS) is difficult because we might update the copies in any order. It is necessary to thus add restrictions to the process of writing such that the WS axiom is not violated. In addition, it is possible that because there is a single bus, a cache might get continuously denied access to the bus, and thus it might not get a chance to let other caches know about a write request that it has received. This will violate the write propagation (WP) axiom. There is thus a need to ensure some fairness such that the WP axiom is not violated – a write is ultimately visible.

Reads

Let us outline a simple protocol.

Whenever a cache receives a read request, if there is a read hit, we are sure it is the correct value, and thus we quickly forward the value to the requester. However, if there is a read miss, then there is a need to search for the value in other sister caches first. Recall that in a conventional system, we would have sent the request to the lower level. In this case, we will not do that. We will first ask other sister caches. Only if all of them indicate that they do not have the block, then only we send the request to the lower level. To send a request to the rest of the sister caches, the cache that has a read miss needs to first get control of the bus. Once it has exclusive access to the bus, it needs to broadcast a read miss request – denoted as RdX. We assume a bus controller that gets requests from different caches, and then allocates the bus to them fairly. This ensures that our protocol follows the WP axiom.

After the cache broadcasts the read miss message, the rest of the sister caches get the message by snooping on the bus. If any of the sister caches has a copy of the block, then it sends it over the bus to the requesting cache. There is a subtle point that has to be made here. Assume three sister caches have a copy of the block. It should not be the case that all three of them send back a copy of the block. This will not happen in a bus based system, because the cache that gets control of the bus first will send a copy of the block. The rest of the caches will see this and decide not to send a response (with a copy of the block) to the requester.

Let us thus propose a simple protocol known as the MSI protocol to implement this high-level idea. In this protocol, each cache line has three states: modified (M), shared (S), and invalid (I). The protocol is as follows. When a given cache line is empty, it is said to be in the invalid state I. When it gets a copy of the block after a read miss, it transitions from the I to the S state. S refers to a shared state, where it is known that the block is possibly shared with other caches. This means that other sister caches might have the same copy of the block with them. This part of the protocol is shown with a state machine

in Figure 9.34. The standard method of annotating a state transition is to create an event-action pair separated with the '|' symbol. For example, the notation "$Evict \mid -$" means that whenever we need to evict a block, we just evict it and do not do anything else. However, if we need to read a cache line, when it is in the invalid state (block not present in the cache), we send a read miss message (RdX) on the bus. Once the block arrives, we transition to the shared state. The action in this case always means that a message is sent on the bus, which every sister cache can read.

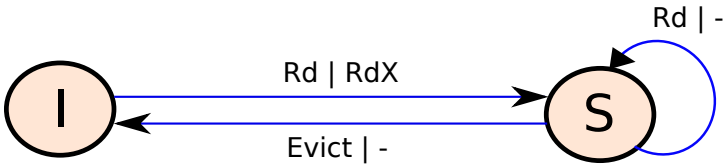

Figure 9.34: State transitions in the write-update protocol: S and I states

Note the transitions from the shared state. If we need to read a line that is already in the shared state, then we can just go ahead and read it. There is no need to send a message to any sister cache. This transition is shown as "$Rd \mid -$" in Figure 9.34.

Next, consider evictions from the cache when the line is in the S state. The S state basically means that we have not written to the block. We are only reading it. Since the current cache has not modified the block, it can seamlessly evict the block. We will not lose any data.

Writes

Let us now consider the tricky case of writes. Assume that we have a write miss. This means that the block is not present in the cache. We need to first request the rest of the sister caches for a copy of the block, and the write can be effected only after we get a copy of the block (like regular caches). This is similar to the case of a read miss. We send a write miss message WrX to the rest of the caches. If we do not get a reply within a specific period of time, or we get a negative response from the sister caches, then it means that the block must be fetched from the lower level. This part is exactly similar to the case of a read miss. Once we get a copy of the block, we transition to the modified (M) state. This is shown in Figure 9.35. Note that till this point we have not performed any read or write yet, we are merely requesting a copy of the block from other caches. Once, we have made the $I \rightarrow M$ transition after receiving a copy of the block, we can then proceed with the read or write operation.

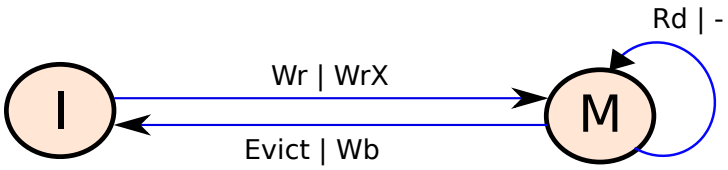

Figure 9.35: The $I \rightarrow M$ and $M \rightarrow I$ transitions (writes in the M state are not shown)

Once, the block is in the M state, reads are seamless. This is indicated by $Rd \mid -$ in Figure 9.35, which means that no message needs to be sent on the bus. We can go ahead and read a copy of the block. However, if we evict the block we need to write a copy to the lower level, because we have modified its contents. If we do not write back a copy to the lower level, then it is possible that we might lose our updates because no other sister cache may have a copy of the block. Hence, to be on the safer side, every time we evict a block in the M state, we should write back the block to the lower level such that updates are not lost. We use the term Wb in Figure 9.35 to denote a write back. We can thus conclude that unlike the S state, in the M state, evictions are more expensive.

Let us now look at writes in the M state, which is the only event-action pair that is not shown in Figure 9.35. This is where we need to ensure that the WS (write serialization) axiom is not violated.

Let us first consider a simple solution that might appear intuitive yet is wrong. Our algorithm could be that we write to the block, and then broadcast the write on the bus to the rest of the sister caches such that they can update the copy of the block that they may have with them. This will ensure that at all points of time, all the caches have the same contents of the block. However, this is not correct. Assume cache C_1 writes 1 to x, and at the same time cache C_2 writes 2 to x. Then they will try to broadcast the values. Whoever (C_1 or C_2) gets control of the bus the last will end up writing the final value. Assume C_1 broadcasts first and then C_2 broadcasts. It is perfectly possible that on C_1 two successive reads will read the following values: first 1 and then 2. Now, on C_2 we might have a read operation that arrives after we have just written 2 to x, and not performed or received any write messages on the bus. In this case, we will read $x = 2$. After C_2 receives the broadcast from C_1 and updates x to 1, the second read operation on x by C_2 will return 1. Thus, the order of writes perceived by C_2 is (2,1), whereas for C_1 it is (1,2). This clearly violates the WS axiom. The writes are not serialized.

Let us thus try to fix this. Let us make writes atomic, where a write is visible to all the caches at the same time. This can be done very easily. When we need to write to a block, which is present in the cache, we wait to get control of the bus. Then we broadcast the write. All the sister caches **update the copy of the block** if they have a copy of it. This includes the requesting cache as well, which effects the write when it is broadcasted successfully. This process ensures that we can implement an atomic write operation, where all the caches see it at the same time. This will ensure write serialization.

Figure 9.36 shows the final diagram for the transitions in the M state. The flow of actions on a write miss is as follows: broadcast a WrX message on the bus, wait to receive a copy of the block, transition to the M state, and then effect the write.

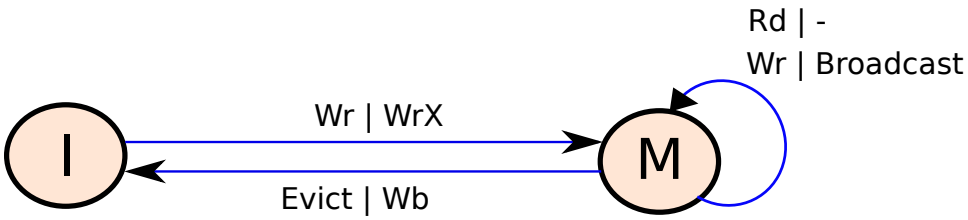

Figure 9.36: $I \rightarrow M$ and $M \rightarrow I$ transitions

Till now, we have looked at transitions from the I state to the S and M states. We now need to look at transitions between the S and M states. In the write-update protocol, we never make a transition from the M state to the S state. This is because if we have write access to the block, then it automatically implies that we have read access to the block. However, we do need to make a transition from the S state to the M state if there is a write request. This would indicate that we have modified a copy of the block. The final state diagram for the protocol with all the three states is shown in Figure 9.37.

Transitions due to events received from the bus

There are three kinds of events that a cache can receive from the bus: RdX (read miss), WrX (write miss), and $Broadcast$ (a write being broadcasted to the rest of the caches). We need to process these messages for each of our valid states: S and M. The state transition diagram is shown in Figure 9.38.

Whenever we get a read miss or a write miss from the bus, it means that some other sister cache needs to be sent a copy of the block. One of the caches that contains a copy of the block needs to reply with the copy, and when the rest of the caches see the reply they need not send their own copies to the requesting cache. In terms of messages, when we receive the RdX and WrX messages from the bus, we need to start the process of sending a copy of the block over the bus to the requesting cache. This is denoted by the $Send$ action in Figure 9.38. The fact that the $Send$ action may be suppressed because a sister cache already sent a copy of the block is not shown in the figure.

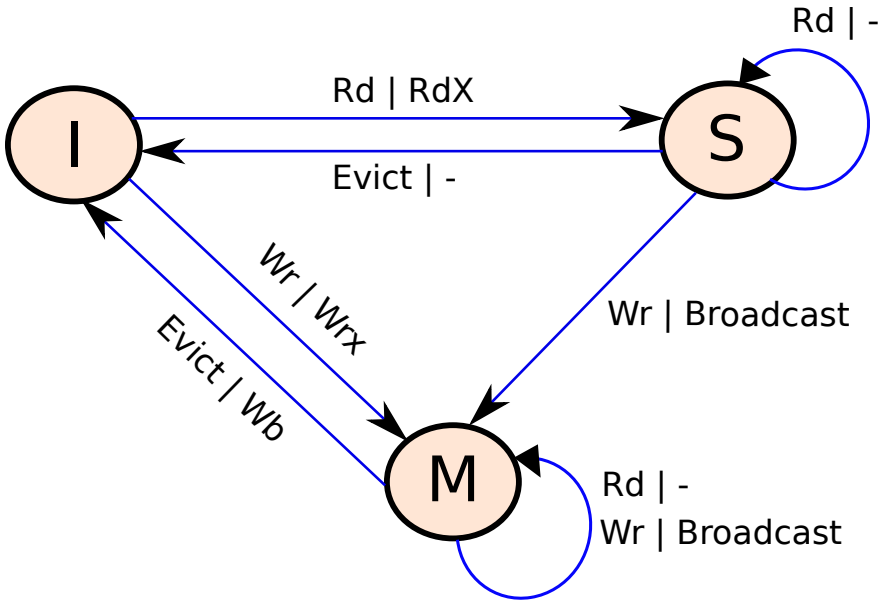

Figure 9.37: The write-update protocol: state transitions due to read, write, and evict events

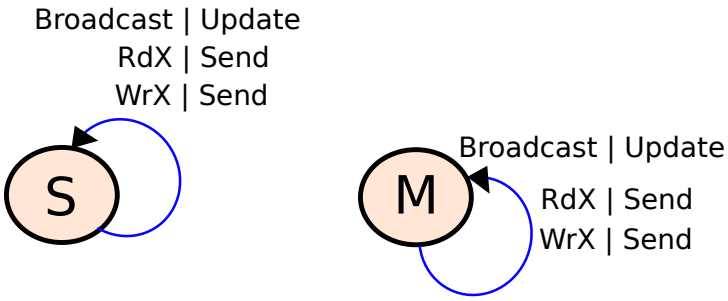

Figure 9.38: The write-update protocol: state transitions due to events received from the bus

The other message that a cache can receive is a *Broadcast* message, which means that a sister cache has written to a block. The caches that contain a copy of the block need to update their local copies with the values being sent over the bus. This is an *Update* action.

Summary

The axioms of coherence hold because of the following reasons. The WS axiom holds because all the writes are atomic, and they are instantaneously visible to all the caches. Alternatively, the caches see the same order of writes, which is also the same as the order in which the caches get access to the bus. The WP axiom needs to be guaranteed by the bus master – circuit that controls access to the bus. It needs to ensure that all the caches get access to the bus in finite and bounded time. This way writes will not be lost. Every cache will be able to place its requests on the bus without waiting indefinitely.

Even though we have guaranteed the axioms of coherence, significant performance and power issues still remain mainly because a write is very expensive. Let us elaborate.

1. For every write operation, we need to broadcast the values on the bus. This increases the bus

Term	Meaning
States	
I	Invalid state. This means that the block is not present in the cache.
S	Shared state. The block can be read and evicted seamlessly. However, we cannot write to the block.
M	Modified state. We can read or write to the block.
Cache actions and events	
Rd	Read request
Wr	Write request
$Evict$	Eviction request for a block.
Wb	Write back data to the lower level.
$Update$	Update the copy of the block with values sent on the bus.
Messages on the bus	
RdX	Read miss
WrX	Write miss
$Broadcast$	Broadcast a write on the bus.
$Send$	Send a copy of the block to the requesting cache.

Table 9.3: Glossary of terms used while describing coherence protocols

traffic and power utilization significantly.

2. Whenever a write is broadcast on the bus, every cache needs to check whether it contains the cache block or not. If it does, it needs to update its contents. This increases the contention at every cache and also increases the power utilization significantly.

Before proceeding to discuss more efficient protocols, let us conclude this section by providing a glossary that defines all the terms used for the states and transitions. We shall use the same terminology later as well. Refer to Table 9.3.

9.4.2 Write-Invalidate Protocol using a Bus

The main drawback of the write-update protocol is that we need to send a message on every write. This increases the power utilization and the bus occupancy significantly. The write-invalidate protocol does not suffer from this problem. To keep matters simple, let us keep the model of the system the same: all the caches are connected to each other using a shared bus. Since every cache can snoop on the bus, this is another example of a *snoopy protocol*. This protocol has the same three states albeit with different connotations.

Basic Insights

The insights for this protocol are as follows. The reason that we need to broadcast writes to the rest of the caches is because the rest of the caches need to be kept up to date. Hence, their state needs to be kept updated all the time. The cost of ensuring this is significant, and it encumbers every single write operation. To solve this problem, we need to constrain the process of reading and writing to copies of the same block. Let us thus propose the following set of rules.

Single Writer At any point of time, we can at the most have only a single writer for each block and no readers.

Multiple Readers If no cache has the permission to write to the block, then multiple caches can read the block simultaneously. In other words, we can support multiple readers at a time.

Either we have a *single writer* situation or a *multiple-readers* situation. We never have a case where we have two caches that can write to different copies of the same block simultaneously. We also do not have a case where one cache is writing to the block, and another cache is concurrently reading it. This is very different from the conditions that we had in the case of the write-update protocol. However, because we allow a single writer at a time, we shall show that we can design a protocol where we do not need to send a message after every write operation.

Let us define the term *conflicting access*. Memory instructions A and B that access the same block are said to be conflicting if one of the following conditions is true.

- A is a write and B is a write.
- Or, A is a read and B is a write.
- Or, A is a write and B is a read.

We can express the set of rules (single writer or multiple readers) that we have seen before in another way:

In the write-invalidate protocol, we do not support concurrent and conflicting memory accesses.

Write-Invalidate Protocol: Read, Write, and Evict events

In the write-invalidate protocol, the shared (S) state represents the multiple-readers scenario, and the M state represents the single-writer scenario. The state transitions for read, write, and evict events are shown in Figure 9.39.

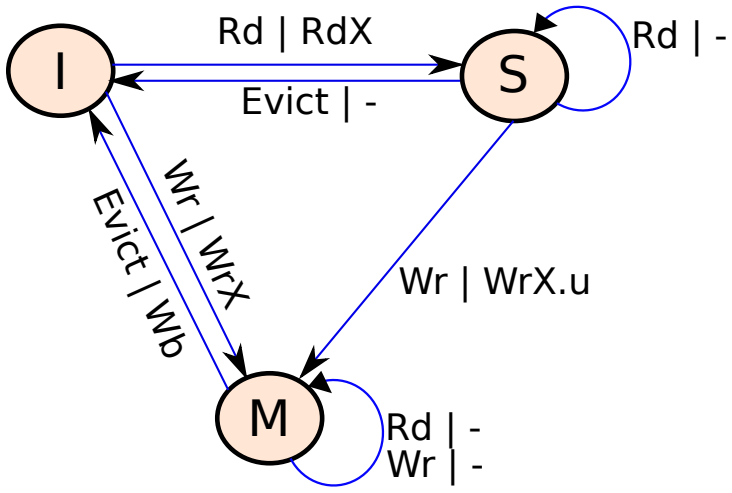

Figure 9.39: MSI protocol (messages from the higher level)

We make a transition from the I state to the S state, when there is a read request. In this case, we send a read miss RdX on the bus, and get a copy of the block. If it is there with a sister cache, then we get that copy, otherwise we get it from the lower level. The logic for avoiding multiple responses is the same as that in the write-update protocol, which is that once a response is sent, the rest of the sister caches that have the data discard their responses.

In the S state, we can read as many times as we want. However, we are not allowed to write to the block. It is necessary to transition to the M state, and for that it is necessary to seek the permission from the rest of the sister caches. Recall that we can only support a single writer at a time. If there is a need to write to the block in the S state, then the cache places a write miss, $WrX.u$, message on the bus. It is important to make a distinction between a regular write miss message WrX, and a write miss upgrade message, $WrX.u$. We have seen the WrX message in the write-update protocol as well. We send a WrX message when the requesting cache does not have a copy of the block. A copy of the block needs to be supplied to it by a sister cache if it is present with it. However, when we are transitioning from the S state to the M state in the write-invalidate protocol, we already have a copy of the block, we do not need one more copy. Instead, we wish to let the sister caches know that they need to discard their copies such that the requesting cache can perform a write. Discarding copies of a block is known as *invalidation*. This is why this protocol is called the write-invalidate protocol. As a result, we send a different message that informs the rest of the caches that the state of the block is being upgraded to the M state. This is why we introduced a new message called the write miss upgrade message $WrX.u$.

Subsequently, we can transition to the M state. In the M state, the cache is guaranteed to have an exclusive copy of the block. No other cache has a copy. The cache is thus free to perform its reads and writes. There is no need to inform the other caches. This is the crux of this protocol. In the M state, a cache can read or write a block any number of times without placing messages on the bus because no sister cache has a copy of the block. This is where we save on messages.

Now consider the $I \rightarrow M$ transition. This happens, when the block is not present in the cache, and we wish to write to the block. In this case, we place a write miss message WrX on the bus. A sister cache, or the lower level forward a copy of the block, and then we directly transition to the M state.

Let us now look at evictions. If a block gets evicted in the S state, then we transition to the I state. Since we were only reading the block, its contents have not been modified. We can thus seamlessly evict the block. Nothing needs to be done. However, an eviction in the M state is more expensive. This is because we have modified the contents of the block, and we are sure that there are no copies of the block in other sister caches. If we seamlessly evict it, then the updates to the block will be lost. It is thus necessary to write-back a copy of the block to the lower level. Then it can be evicted, and we can make an $M \rightarrow I$ transition.

Write-Invalidate Protocol: Messages from the Bus

Let us now see how a cache needs to react to messages coming from the bus (see Figure 9.40). Consider the S and M states because considering the I state in this case is pointless. In the S state, if we get a read miss message, we can prepare a response with a copy of the block. If no other cache has sent a response already, then the response can be immediately sent. However, if we get a write miss, it is necessary to transition to the I state; recall that only one cache can have the permission to write to the block at any point of time.

In the M state, if we receive a read miss, then it means that another cache is requesting for read-only access. Since it is not possible for two copies of the block to be in the M state simultaneously, or for one copy to be in the M state and the other copy to be in the S state, we need to do several things.

1. Provide a copy of the block. This needs to be done because no other cache, or even the lower level of memory contains an up-to-date copy of the block.
2. Transition to the S state ($M \rightarrow S$). The cache will at least be able to seamlessly read the data in the future.
3. Write back a copy of the block to the lower level. This needs to be done because we wish to have seamless evictions in the S state. Assume that we make an $M \rightarrow S$ transition, and we do not write-back a copy. Subsequently, if the copies get evicted, the updates will be lost, because we support seamless evictions from the S state. Hence, a write-back is required upon an $M \rightarrow S$ transition.

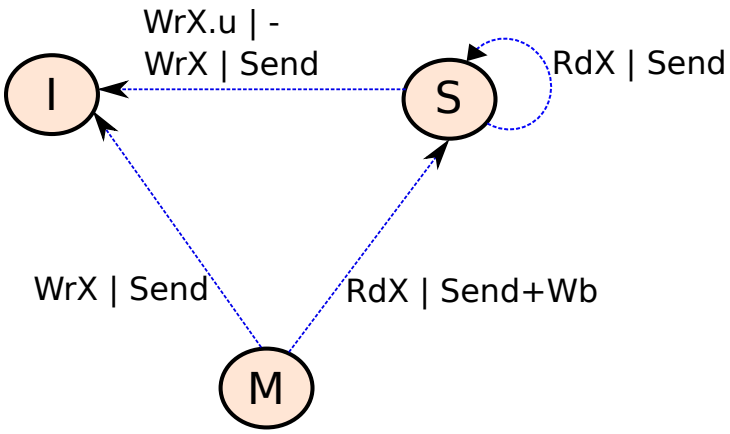

Figure 9.40: MSI Protocol with invalidate messages (messages from the bus)

Let us now consider write miss messages. In the S state, if we get a write miss message then it means that we need to transition to the I state. This is because when a cache writes to a block, it needs to do so exclusively. If the message is WrX, we need to forward a copy of the block to the remote cache because it does not have a copy. However, if the message received is of type $WrX.u$, which is just an announcement that the requesting cache is transitioning the state of the cache line containing the block from the S to the M state, we do not have to send a copy of the block. The requester already has a copy of the block in the S state. On the same lines, we need to transition to the I state from the M state upon receiving a write miss. Since the block was in the M state, the remote cache does not have a copy of it; it is thus necessary to forward a copy.

Summary of the Write-Invalidate Protocol

Let us summarize our discussion. The write-invalidate protocol either allows a single writer or multiple readers to simultaneously exist for each cache block at any single point of time. Since we do not have multiple writers at a time, the order of writes to a single location can be clearly established. It is the order in which we enter the M state across the caches. This ensures that the WS axiom holds. For the WP axiom, we need to ensure that if a cache needs to write to a block, it ultimately gets write access to it. This is possible to ensure by having a fair bus that gives every cache a fair chance to send messages on the bus. Once a waiting cache gets access to the bus, it can send a write miss message on the bus and subsequently get write access to the block. Note that some corner cases are possible. For example, it is possible that before the cache is able to write the data, it receives a write miss on the bus from a sister cache. In such cases, the pending operation – write operation in this case – needs to complete first. Such policies will ensure that the axioms of coherence hold.

9.4.3 MESI Protocol

Let us now make the snoopy protocol slightly more efficient. Assume that a cache has a read miss, and it tries to read the data from other sister caches. If none of the sister caches have the block, then there is a need to go to the lower level in the memory hierarchy. Once the block is fetched, the cache is sure that it holds an exclusive copy of the block, and no other sister cache has a copy of the block. Now, if it desires to write to the block it needs to follow the same procedure that entails broadcasting a write miss on the bus, and waiting for other caches to invalidate their copies. This is not required, given the fact that we already know that no other cache has a copy of the block. The MSI protocol has no way of dealing with the situation. It will always broadcast a message when we need to transition from the S to

the M state. This can be fixed by adding an extra state called the exclusive state – E state. This state will indicate that the given cache can read the block and no other sister cache has a copy.

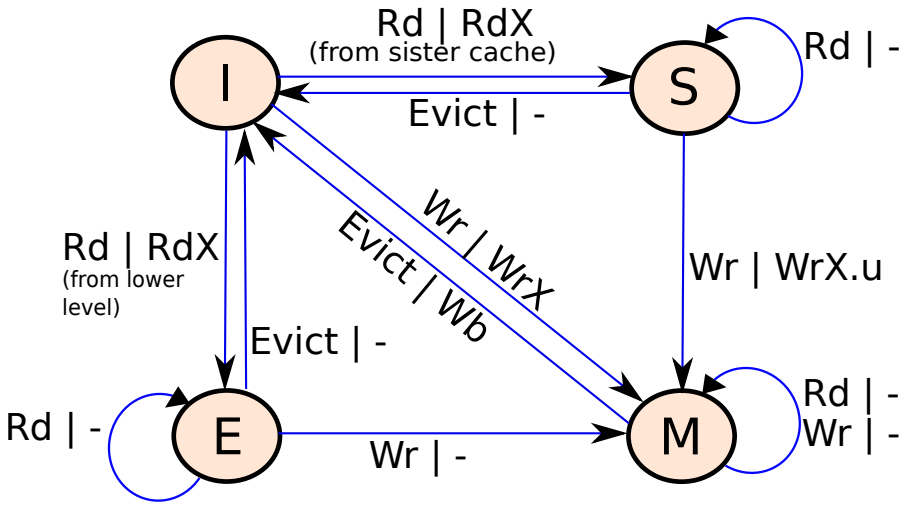

Figure 9.41: MESI protocol (read, write, and evict)

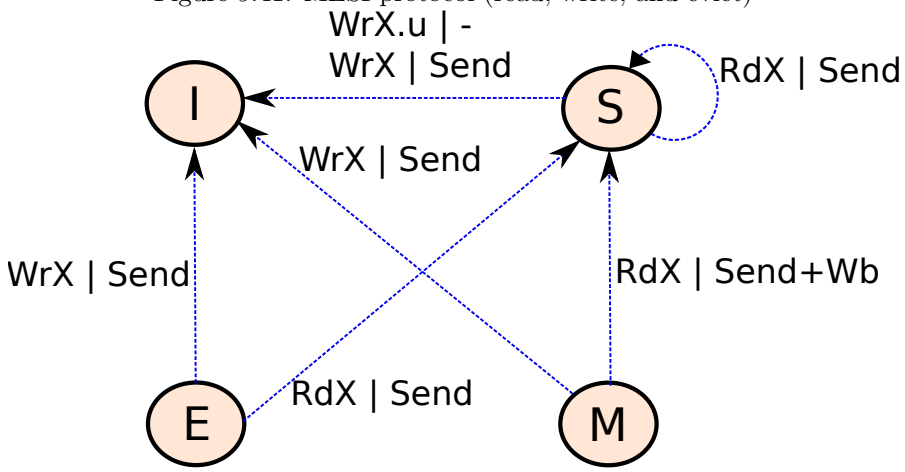

Figure 9.42: MESI protocol (bus events)

Let us explain the E state in the context of the state diagram that only considers reads, writes, and evicts. Refer to Figure 9.41. Let us start from the I state. If we have a read miss (denoted by Rd), there are two possible choices. Either the data is present in a sister cache, or we need to fetch it from the lower level. Initially we do not know. We place an RdX message on the bus. If we get a reply from a sister cache, then we transition to the S state, because we know that there are multiple copies of the block. However, if we do not get a reply from any sister cache, then it is necessary to fetch the block from the lower level. After fetching a copy, we can transition to the E (exclusive) state because we are sure that no other cache has a copy.

The S and M states behave in the same manner as the original MSI protocol. Let us thus solely focus our attention on the E state. We can seamlessly read a block that is in the E state because no other cache is writing to it. However, the key advantage of having the E state is that we can silently move from the E to the M state if we need to write to the block. There is no need to send a message on the bus. This is because there is no need to send any invalidate messages to any of the other sister

caches (they do not have a copy of the block). Eviction from the E state is also seamless (no messages are sent) because the data has not been modified.

Let us now look at the state transitions due to events received from the bus (see Figure 9.42). The transitions for the M, S, and I states remain the same. Let us divert our attention to the transitions from the E state. We can receive two kinds of messages from the bus: RdX and WrX. RdX indicates that another sister cache wants to read the data. In this case, we need to send the data and then transition to the shared state. This is because at this point of time two caches contain a copy of the block; it is not *exclusive* to any single cache. Second, if we get a write miss message, WrX, on the bus, then we need to make an $E \rightarrow I$ transition and also send the data to the requester. This transition is similar to the $M \rightarrow I$ transition.

Let us summarize.

1. The MESI protocol adds an extra E (exclusive) state. The state transitions for the M, S, and I states remain mostly the same.
2. The main advantage of the E state is that we can take advantage of codes where we access a lot of blocks that are not shared across the caches. In such cases, we should have the ability to silently write to the block without sending invalidate messages on the bus. The E state allows us this flexibility.

The MESI protocol reduces the traffic on the bus as compared to the MSI protocol. However, both the protocols suffer from the same problem, which is that we need to perform frequent write-backs to the lower level, when we have a transition from the M to the S state. Note that this is a very frequent pattern for shared data. Write-backs to the lower level are required to ensure that we can perform seamless evictions from the S state. Let us try to fix this issue by introducing one more state, where the explicit aim is to reduce the number of write-backs to the lower level.

In addition, we need to solve one more problem. Whenever we have a read miss or a write miss, a sister cache needs to forward a copy of the block. If multiple caches have a copy, then all of them will try to send a copy; however, we want only one of them to succeed. Our current solution is that all of them create their responses, and the moment they see a response on the bus sent by a sister cache, they discard their responses. This is time consuming, and requires additional hardware support. It is possible to do something better such that most of the sister caches do not create such responses in the first place. The process of choosing one candidate among a set of interested candidates, like sister caches in this case, is known as *arbitration*; our aim is to ease this process or eliminate its need by proposing a more efficient cache coherence protocol.

Definition 75
The process of choosing one entity among a plurality of interested entities (software or hardware) is known as arbitration*. For example, in this case, multiple sister caches compete among each other to send a response to the requesting cache. There is thus a need for arbitration.*

9.4.4 MOESI Protocol

We need to solve two problems:

1. Minimize the number of write-back messages that write data back to the lower level. These messages are slow and time consuming.
2. Eliminate (as far as possible) the need for arbitration while forwarding a copy of the block to the requesting cache.

We shall achieve this by creating an additional state called the owner, O, state, and two more temporary states – St and Se. If a cache contains a block in the owner state then it is by default responsible to forward the data. This ensures that caches do not compete among each other to supply data to the requesting cache. Furthermore, caches do not have to prepare responses, and discard them. A lot of effort will get reduced by just adding this extra state. In addition, the O state can contain data that has been modified. The aim is to eliminate write-backs as far as possible.

Let us thus create a MOESI protocol, where we have the MESI states, and an additional owner state. The state transition diagrams are shown in Figures 9.43 and 9.44 for messages received from the bus and regular read/write/evict events respectively. We shall argue later that we need the two temporary states St and Se for the sake of correctness.

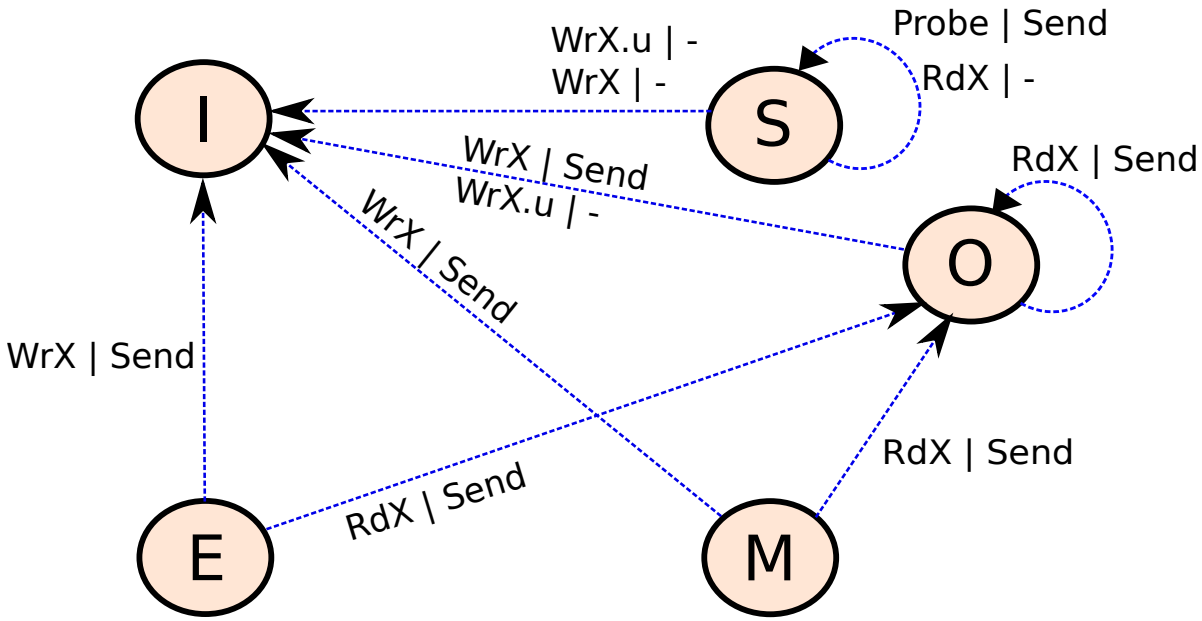

Figure 9.43: MOESI protocol (bus events)

Let us first focus on Figure 9.43 that shows the state transitions after receiving a message on the bus. We can transition to the O state from either the E state or the M state. Whenever a block is in the E state and an RdX (read miss) message is received from the bus, it means that another sister cache is interested in reading the block. In the MESI protocol, we would have transitioned to the S state. However, in this case, we set the new state as the O state. After this operation, there are two caches that contain a copy of the block: one has the block in the S state and the other has it in the O state. Henceforth, if another sister cache has a read miss and requests for a copy of the block, arbitration is not required. The cache that has the block in the S state simply ignores the read miss message. The only cache that responds is the one that has the block in the O state. It responds with the contents of the block. Thus, there is no need for arbitration.

The other interesting feature of the O state can be observed by taking a look at the $M \rightarrow O$ transition. We make an $M \rightarrow O$ transition when we receive an RdX message on the bus (instead of transitioning to the S state). The cache with the block in the O state subsequently keeps supplying data to requesting caches. Note that in this case, the block's contents are possibly modified, yet we do not perform a write-back. If another cache wishes to write to a block by sending a WrX (write miss) or $WrX.u$ (write upgrade) request, then the block simply transitions from the O to the I state. In the former case, there is a need to send the contents of the block; however, in the latter case, there is no need to send a copy.

We add one more message called the *Probe* message. If in the S state, a *Probe* message is received, then we send a copy of the block. The reasons will be clear later. The rest of the transitions remain the same.

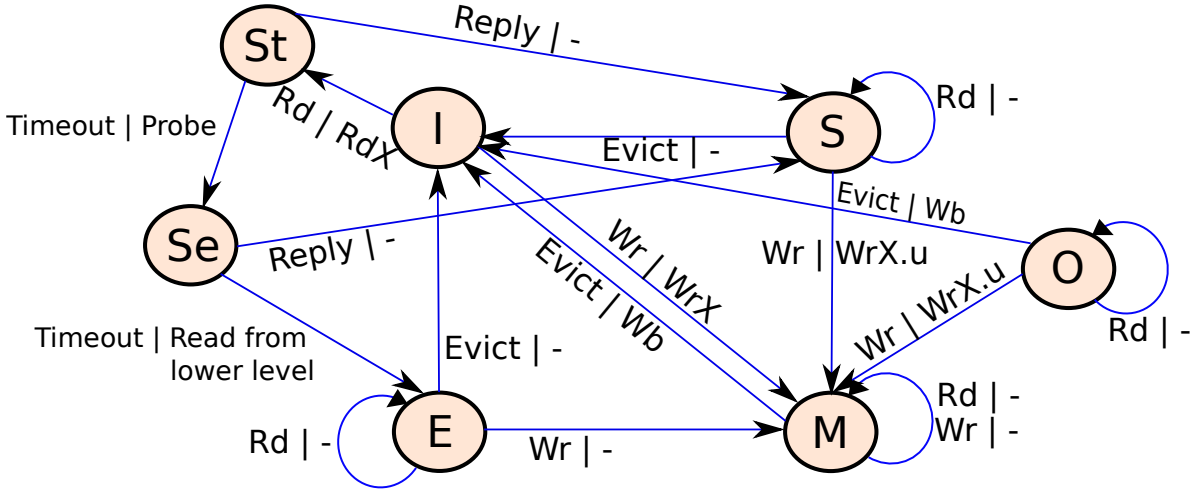

Figure 9.44: MOESI protocol (read, write, and evict)

Let us now look at regular reads, writes, and evict operations (see Figure 9.44). The transitions in the M, E, S, and I states are the same, other than the $I \rightarrow S$ and $I \rightarrow E$ transitions; they require new intermediate states, which we shall discuss later. The main addition is the O state. In the O state, we can seamlessly read data. However, we are not allowed to write to the block. In this case, there is a need to invalidate the rest of the copies by sending a write upgrade message $WrX.u$, and to transition to the M state. In the M state, we can seamlessly read and write to the block.

Let us now consider the case in which we evict a block in the O state. Since the O state can possibly contain modified data, we need to write the data to the lower level. A write-back is obvious in the $M \rightarrow I$ transition because no other cache has a copy of the block. However, in an $O \rightarrow I$ transition because of an eviction, there is theoretically no need to perform a write-back if another cache has a copy.

There is space for a new optimization here. If a sister cache has a copy of the block, then we need not write the data to the lower level upon an eviction in the O state. We should simply transfer ownership. However, this requires arbitration because the block might be present with multiple sister caches. This overhead is justified if it is significantly lower than accessing the lower level. The MESIF protocol that introduces a new F state (on the lines of our O state with some differences) has the notion of transfer of ownership. We can introduce this notion in our MOESI protocol as well. The reader is advised to look up the details of the MESIF protocol on the web. For the sake of simplicity, let us not introduce this state and continue without it.

In the vanilla MOESI protocol, we do not transfer ownership, instead we write-back the block upon an eviction in the O state. Assume that there are other caches that have a copy of the block in the S state. Now if a new cache requests for a copy of the block, it will not find a cache that has the block in the O state. It will thus be forced to read a copy of the block from the lower level, which is inefficient. There is a correctness problem as well. When a cache that does not contain a copy of the block reads the block from the lower level, what is its state: E (exclusive) or S (shared)? We do not know if sister caches have a copy or not. Either they have the block in the S state only, or none of them have a copy. We have no way of distinguishing between these two situations.

Hence, we introduce two temporary states: St and Se (see Figure 9.44). Let us focus on the tricky corner case when there is a read miss from the I state. We send a read miss (RdX) message on the bus

and transition to the St state. This is a temporary state, because we need to make a transition from it in finite time. Then we wait for a reply. If we get a reply from a sister cache (*Reply* in the figure), we transition to the S state. Otherwise, we wait till there is a timeout. We assume that there is a timeout period after which we can conclude that none of the caches have a copy of the block in the O, E, or M states. After a timeout, we transition to the second temporary state, Se, and simultaneously we send a *Probe* message on the bus.

If any cache has the block in the shared state, it prepares a response with a copy of the block. The cache that gets control of the bus first sends the response. Once we get this message (*Reply* in the figure), we can transition to the S state because another sister cache also has a copy of the block. However, if we have a timeout in the Se state, we can conclude that no sister cache has a copy of the block in any state. We thus need to read the block from the lower level. We do this, and then transition to the E state because we are sure that no other cache has a copy of the block.

To summarize, we observe that even though the MOESI protocol solves an important problem by introducing an additional state, there is a need to add two temporary states to solve resultant correctness problems.

9.4.5 Write-Invalidate Protocol using a Directory

Let us now try to make the write-invalidate protocol even more efficient and scalable. Recall that we proposed the invalidate protocol in the place of the update protocol to reduce the number of bus accesses; we wanted to reduce the contention on the bus. Let us go one step forward and try to improve the performance even more.

The biggest problem with bus based protocols is the bus itself. The bus is by definition a centralized structure and can only handle one message at a time. It is true that a protocol using a bus naturally places an order on all the requests based on the order in which they get access to the bus. This proves beneficial while ensuring the axioms of coherence. However, at the same time, it reduces the communication bandwidth, and thus does not scale with the number of constituent caches. For 2-4 caches, using a bus is a good idea. However, as we increase the number of constituent caches, the bus fails to scale. We need to use a network-on-chip (NoC) as we studied in Chapter 8. In an NoC we can sustain many parallel read-write operations between pairs of nodes, and thus the net bandwidth is much larger.

Sadly, we lose the most important advantage of a bus with regard to the cache coherence axioms – ensuring an order between write operations. Ensuring the write propagation axiom is still easy because we can always design an underlying network that provides fairness guarantees.

To solve these issues, let us *centralize* our NoC. This means let us add a new node in the NoC whose job is to provide an order between write operations – *serialization*. Let us refer to this structure as the *directory*. It provides serialization along with a few more services to the set of caches in a distributed cache. The conceptual diagram of the system is shown in Figure 9.45. A cache coherence protocol that uses the directory is known as a directory protocol.

Structure of a Directory

The main role of a directory is to keep track of the sharing status of all the blocks in a cache. It is organized as a cache where we have a standard tag array and a data array. It contains a list of entries known as directory entries, where an entry corresponds to a single block. The structure of a directory entry is shown in Figure 9.46. We have a state field that stores the state of the block, and then we have a list of sharers. The state indicates if the block is shared, or held exclusively by a single cache. The list of sharers is a list of cache ids that contain copies of the block.

The state field is similar to the state fields that we maintain in the MSI based protocols that we have already seen. However, unlike the snoopy protocols where we never maintain a list of caches that contain a copy of the block, in this case we need to maintain an explicit list. Since we do not have a bus

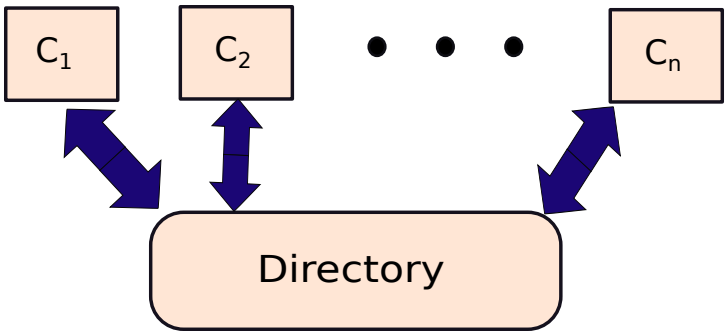

Figure 9.45: Conceptual view of a directory. $C_1 \ldots C_n$ are the sister caches connected to the directory via the NoC.

Figure 9.46: Structure of a directory entry. The tag part of the block address is not shown.

based configuration, a broadcast is a very expensive proposition in an NoC, and thus it is necessary to maintain an explicit list and send point-to-point messages the *sharers* of a block. A *sharer* is a sister cache that has a copy of the block.

The simplest way for storing a list of sharers is to have a bit vector where each bit corresponds to a sister cache. If the i^{th} bit is set, then it means that the i^{th} sister cache has a copy of the block. If there are N sister caches in a given distributed cache, then each entry in its directory contains N bits (1 bit per cache). This is known as the *fully mapped* scheme. We shall discuss more schemes to optimize the list of sharers after discussing the cache coherence protocol.

Definition 76
A scheme in which we have an entry for each sister cache in the list of sharers is known as a fully mapped scheme.

Protocol

Let us design a MESI protocol. Let us keep the same set of states: M, E, S, and I at the level of each constituent cache. The only difference is that whenever we evict a block we need to send an *Evict* message to the directory. We were not doing this in the case of the snoopy protocols. Moreover, all the read and write miss messages are sent to the directory first, not to the sister caches.

Taking this account, let us list the messages that a cache sends on the NoC. They are as follows: RdX (read miss), WrX (write miss), $WrX.u$ (write upgrade), and *Evict* (block evicted from a cache). The rest of the state transition diagram for read, write, and evict messages remains the same. Hence, we do not show the modified state diagrams for these events. Let us instead focus on the **state transition of a given directory entry** as shown in Figures 9.47 and 9.48. We use three states: U (uncached), E (a sister cache contains a copy of the block in either the E or M state), and S (one or more sister caches have the block in the S state).

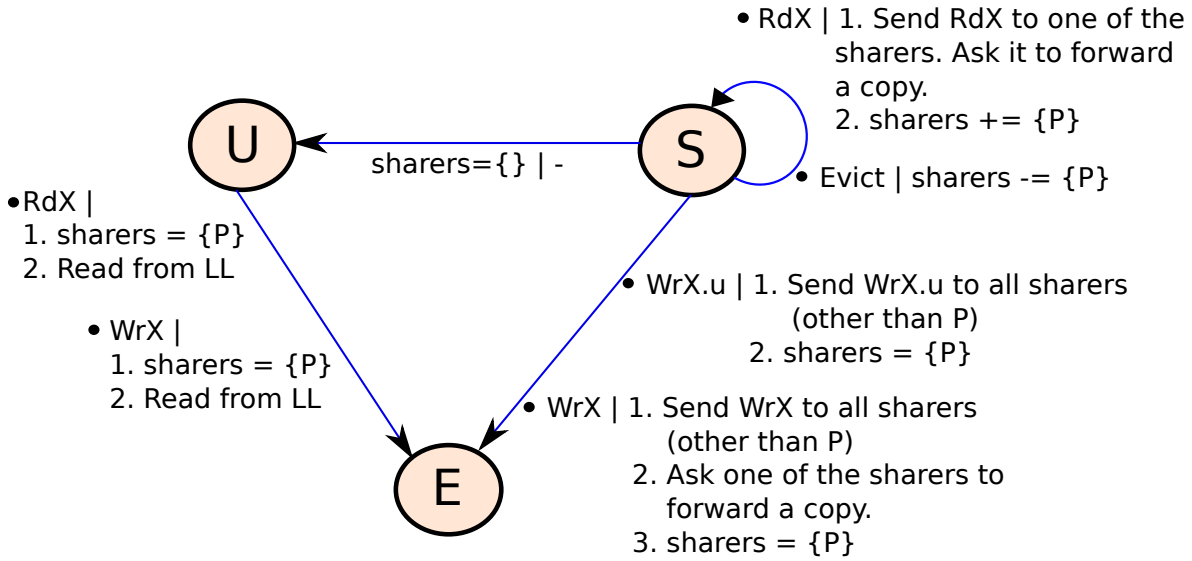

Figure 9.47: State transitions in a directory entry (from the U and S states)

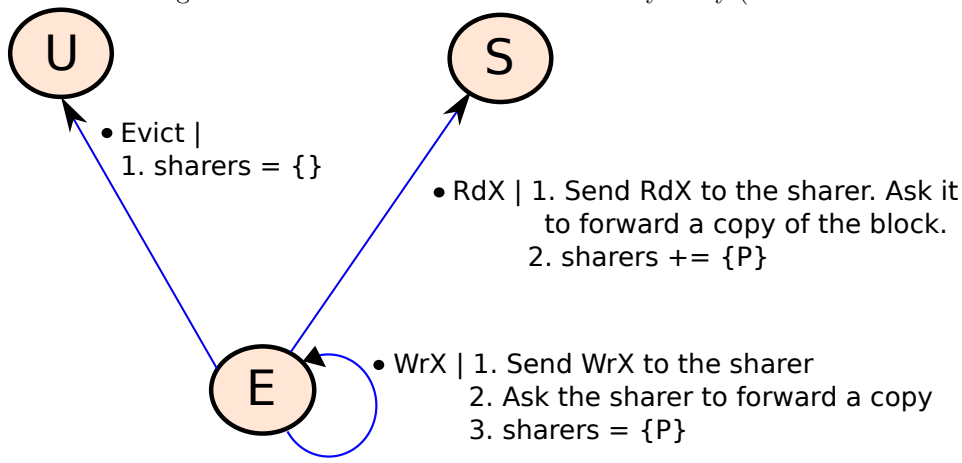

Figure 9.48: State transitions in a directory entry (from the E state)

Let us focus on Figure 9.47 that shows all the transitions from the U and S states. Initially, we start from the uncached (U) state. In this state there are no sharers – no sister cache contains a copy of the data. Whenever the directory gets an RdX message from a cache, it transitions to the E (exclusive) state: only one sister cache contains a copy of the block in either the E or M state. For this $U \rightarrow E$ transition, the directory can initiate a read from the lower level (LL in the figure) to get a copy of the block and forward it to the requesting cache. Let us adopt a convention to designate the id of the sister cache that is sending an event to the directory as P. In this case we add P to the list of sharers, which hitherto was empty.

We have an S state, which has the same connotation as the S state in the MESI protocol. It represents the situation where the block is present in one or more sister caches in the shared state. In the S state, we can keep on receiving and responding to read miss (RdX) messages from caches. In each case, we forward the read miss message to one of the sharers, and ask it to directly send a copy of the block to the requesting cache. The response need not be routed through the directory. Subsequently, we add P

to the list of sharers. If we get an evict message, then we remove P from the list of sharers. If the list of sharers becomes empty, then it means that a copy of the block is not present in any sister cache, and we can thus change the state to U.

Next, let us consider the write miss messages: WrX and $WrX.u$. We need to transition to the E state. The E state indicates that only one cache contains a copy of the block, and the block in that cache can either be there in the E state or M state. Recall that $E \rightarrow M$ transitions are silent, and thus the directory will never get to know if the block has transitioned from the E to the M state in a sister cache. Hence, we have just one state to denote exclusivity with possible write access in the directory entry. First consider the $U \rightarrow E$ transition. This happens when we get a WrX message from a sister cache. In this case, there is a need to read a copy of the block from the lower level because no other cache contains a copy, and forward it to the requesting cache P. In addition, we make P the only sharer because it has an exclusive copy of the block. We transition from the S to the E state upon receiving two kinds of messages: WrX and $WrX.u$. In the case of the upgrade message $WrX.u$, the requesting cache already has a copy of the block. It is just requesting for write permission. We thus need to send invalidate messages ($WrX.u$) to all the sharers other than P, and make P the only sharer. In the case of the WrX message, it means that the requesting cache does not have a copy of the block. Hence, it is necessary to additionally ask one of the existing sharers to forward a copy of the block to the requesting cache P.

Now consider Figure 9.48, which shows the transitions from the E state. Upon an eviction, the list of sharers will become empty, and we need to transition to the U state. This is because no sister cache will contain a copy of the block after the block is evicted. If there is a read miss (RdX), then we make an $E \rightarrow S$ transition. Additionally, we send an RdX message to the lone sharer such that it can move to the S state, and also provide a copy of the block. The requesting cache, P, is then added to the list of sharers.

For a write miss (WrX) message, we do not need to change the state. The state can remain to be E. However, we need to invalidate the sharer, forward the requesting cache a copy of the block, and update the list of sharers to contain only the requesting cache P.

Let us summarize. A directory has taken the place of a snoopy bus. It acts as a point of serialization where the order of writes is determined by the order in which the directory chooses to process them. This ensures the WS axiom. To ensure the WP axiom, it is necessary to ensure that the directory is fair – it does not delay write requests indefinitely. This can easily be achieved with a FIFO queue.

Let us now look at some optimizations and consider corner cases.

9.4.6 Optimizations and Corner Cases in the Directory Protocol

Evicting an Entry from the Directory

The directory needs to have a finite size. It cannot have an entry for every single block in the memory system. If we have 32 GB of main memory, and each block is 64 bytes, then we need half a billion entries, which is clearly not practical. The directory is thus organized as a set associative cache, where each way contains directory entries. When we access the directory, we first search for the entry, and if there is no entry for the block address, then it is necessary to allocate a new entry.

Let us now consider the case when an entry needs to be evicted. Note that we need to evict the state of the block (stored in the directory entry) and the list of sharers. One option is to keep a copy of the directory entries in main memory or in secondary storage. The other option is to discard the contents of the entry altogether after it is evicted. The first option necessitates a lot of storage in the memory system and in secondary storage. It is thus not practical. The only feasible solution is to forget about the contents of the directory entry after it is evicted. However, this will cause problems. We will forget the list of sharers. Next time if the block associated with the entry is accessed, we will not be able to run the state machine correctly because we would have no record of its previous state and the list of sharers.

The only way out of this quagmire is to invalidate all the sharers once a directory entry is evicted. If the block has been modified, then we write it back to the lower level. This ensures that the next time we access the block, the directory entry can be initialized to the pristine U state. Sadly, the process of invalidation and write-back increases the overheads significantly and makes the protocol slow. There is however no choice, and thus our strategy should be to reduce the number of evictions from the directory as far as possible. We thus need a very good replacement algorithm.

Multiple Directories

If we have a single large directory, then we need many read and write ports to cater to different requests every cycle. Thus, we will require a large multiported storage structure in the directory. This will be slow and consume a lot of power. Hence, an effective idea is to split the physical address space into disjoint subsets, and associate a directory with each subset. For example, we can create 8 such subsets by considering the 3 LSB bits of the block address. A subset corresponds to a distinct combination of bits. This way we can create subsets that are mutually disjoint. We can then create 8 separate directories: one for each subset. This is similar to creating a banked cache and the reasons for doing this are the same. Each directory will be smaller, and hence faster; additionally, it will also suffer from less contention. Note that there is no correctness issue here.

Managing the List of Sharers

Let us now consider the list of sharers. In a large server processor we can have tens of cores. Often it is possible to add more processors and cores using expandable slots in the motherboard at runtime. If we were to design for the worst case, then we have to create space for the maximum possible number of sharers, which is the maximum possible number of cores in the system if we are considering a distributed L1 cache. In large servers, this can be a fairly large number such as 256. Adding 256 bits to every entry in the directory is a significant overhead. This should thus be avoided.

Thankfully we can leverage some patterns here. It is very unlikely that a single block will be accessed by all the threads running on all the cores. The degree of sharing is limited to 4 or 8 sister caches in an overwhelming majority of cases. Thus instead of having a bit-vector based scheme for storing the list of sharers, we can optimize the space by explicitly storing a few sharers: 4 or 8. If the maximum possible number of sharers is 256, then it takes 8 bits to represent each sharer; therefore, we need to store 32 or 64 bits in each directory entry depending on the number of sharers that we wish to support. Note that storing 32 or 64 bits as compared to 256 bits is a significant reduction in the number of bits that need to be stored. Hence, a significant savings in storage space is possible. Such a scheme is known as a *partially mapped* scheme.

Definition 77
A partially mapped scheme in a directory refers to a method where we explicitly store the ids of a limited number of sister caches in the list of sharers. We do not have a dedicated entry in the list of sharers for every single sister cache in the ensemble of caches.

If the number of sharers is less than the maximum number of entries that we can store, then there is no problem. However, if the number of sharers exceeds this number, then there is an overflow. We did not have this problem in a fully mapped scheme, where we had a single bit for every sister cache. However, in the partially mapped scheme, we shall have the problem of overflows. There are several strategies to deal with this situation.

Replace: Assume that a directory entry can store up till K sharers. If it is full, and we need to add an additional entry to the list of sharers, we have the problem of overflow. In this case, we select one

of the K sharers and invalidate its contents. It does not remain a sharer anymore. In its place, we can store the id of the requesting cache for the current request.

Invalidate All: The other option is to have an overflow bit. This bit indicates that we were not able to fit the ids of all the sharers in the list of sharers, owing to space constraints. If the overflow bit is 1, then it means that we have had an overflow. This is not a problem for read accesses; however, it is a problem for write accesses because every single copy needs to be invalidated. The most feasible solution in this space is to send invalidate messages to all the sister caches that are a part of the distributed cache after receiving a write miss. This is undoubtedly a slow and time consuming process.

Coarse Grained Coherence: Another solution is to change the granularity of the information. Assume that we store the ids of 8 caches in a system with 256 caches. In this case, the id of each cache is 8 bits long, and since the list of sharers stores 8 such ids, it needs 64 bits of space. Now assume that a block is present in 9 caches. This situation represents an overflow. Let us change the granularity of information that is stored. Let us divide the set of 256 caches into 128 sets that contain 2 caches each (with consecutive ids). Since we have 128 sets, we require 7 bits to uniquely identify each set. In these 64 bits, let us store the ids of 9 such sets containing two caches each.

The advantage of this scheme is as follows. In this case, 9 sets can potentially cover up to 18 caches. Even in the worst case when we do not have two caches in the same set, we can still cover 9 caches, which is one more than what we could do before. We can increase the granularity of this scheme further and cover more caches. In the worst case, we can have one large set containing 256 caches. The advantage of this scheme is that all the sharers are mapped to at least one set. The disadvantage is that we have no way of recording which caches in a set are genuine sharers and which caches are not. This means that if we need to send an invalidate message, we need to send it to all the caches in a set. Those that have a copy of the block will invalidate it, and the rest of the caches will ignore the message. This adds to the overheads of the scheme.

However, this scheme is very flexible. If a block is stored in a single cache then we use a granularity of 1, and in the worst case if is contained in all the caches, then we use a granularity of 256. We can easily adopt the resolution of our sharing vector (list of sharers) depending on the degree of sharing of a block.

Race Conditions

Till now, we have assumed that the transition between states is atomic: appears to be instantaneous. However, in practice this is not the case. Assume we are transitioning from the S to the M state. In this case, the requesting cache needs to first send a message to the directory, and then wait. The directory in turn needs to first queue the request, and then process it when it is the earliest message for the block. The directory then sends write miss (invalidate) messages to all the sharers, and waits for them to finish their state transitions. In most practical protocols, the sharers send acknowledgements back to the directory indicating that they have transitioned their state. After collecting all the acknowledgements, the directory asks the requesting cache to change its state and perform the write access.

To support this long chain of events, we need to add many more waiting states that indicate that the respective caches and directories are waiting for some message or some event. Furthermore, modern high performance protocols try to break the sequence of actions and interleave them with other requests. These are known as *split transaction* protocols. In an environment with so much complexity, we need to effectively deal with race conditions (concurrent events for the same block).

Let us elaborate. Assume cache A has a block in the modified state. Cache B wishes to read a copy of the block. Cache B sends a message to the directory and the directory initiates the process of getting a copy of the block from A. However, assume that before the messages reach A, it decides to evict the block. In this case, there is a race condition between the read miss and the evict. The relative

ordering of the actions is important. If A evicts the block before a copy is sent to B, then the directory will search for the copy of the block in A, and it will not find it. If it goes to the lower level, here also there is a race condition. We need to ensure that an earlier write-back reaches the lower level before we search for a copy of the block in the lower level later. Since these messages are sent via the NoC, their ordering cannot be guaranteed, and there is a chance that a reordering may happen. One option is that we do not allow A to evict the block till it gets a final confirmation from the directory; this will happen after B completes its operation. Such design choices are overly conservative and restrict performance. To get more performance, protocols typically add more states, transitions, and messages (see the Cray X1 protocol [Abts et al., 2003]) such that we can achieve a better overlap between different operations on different copies of the same block. The main idea behind such protocols is that we add more waiting and pending states where the caches and directories wait for parts of their operations to complete. This adds more states to the protocol and more transitions. It is not uncommon for protocols in modern processors to contain more than 20-30 states and 100+ transitions.

To summarize, while designing correct cache coherence protocols in the presence of simultaneous requests and resultant race conditions, we need to add more states and transitions to a protocol. Vantrease et al. [Vantrease et al., 2011] report the existence of cache coherence protocols that have up to 400 state transitions. Verifying these protocols requires exhaustive testing and massive formal verification efforts. There is a trade-off between correctness and performance, and thus such complex protocols are necessary for performance reasons, even though they require a significant design and verification effort.

False Sharing

Our cache coherence protocol operates at the granularity of cache blocks. A typical cache block is 64 or 128 bytes wide. However, a typical access to memory is for 4 or 8 bytes.

Let us now look at another problem that will only happen with multiprocessor coherence. Consider a block with 64 bytes where the bytes are numbered from 1 to 64. Assume core A is interested in bytes $1 \ldots 4$, and core B is interested in bytes $33 \ldots 36$. In this case, whenever core A writes to bytes $1 \ldots 4$, it will invalidate the copy of the block that is there with B. Similarly, when B writes to bytes $33 \ldots 36$, it will invalidate the copy of the block with A. Even though there is no actual overlap between the data that is accessed by cores A and B, they still end up invalidating copies of the same block in each other's caches. Such a phenomenon is known as *false sharing*. Here, we have cache misses and invalidations because two separate cores are interested in different sets of bytes that are a part of the same block. As opposed to false sharing, we can also have true sharing where invalidations happen because two cores access the same data bytes. In this case, there is a need to genuinely invalidate a copy of the data residing in the other core's cache. False sharing is a consequence of the fact that an entire block is treated as one atomic entity.

Definition 78
When two threads running on separate cores have conflicting accesses for the same set of data bytes, the associated cache lines will keep getting invalidated, and the data block will keep moving between caches. Such cache misses are known as true sharing *misses, because the cause of the misses is data sharing between threads.*

As opposed to true sharing, we can have false sharing*, which is defined as follows. In this case, both the threads make conflicting accesses to disjoint sets of bytes within the same cache block. Note that the sets of data bytes do not have any overlap between them. In spite of this, because of the nature of our coherence protocol that tracks accesses at the level of blocks, we shall still have invalidations, and block migration. This is an additional overhead, and will also lead to an increased number of read and write misses.*

A lot of misses in parallel programs can be attributed to false sharing. The common approaches for handling false sharing are as follows.

1. Use a smart compiler that lays out data in such a way that multiple threads do not make conflicting accesses to the same block. It is necessary for the compiler to find all overlapping accesses between threads, and ensure that data is laid out in such a way in memory that the probability of false sharing is minimized.

2. Use word-level coherence tracking. In this case, we modify the invalidate protocol to allow conflicting accesses to different non-overlapping parts of a block. We maintain multiple copies and explicitly keep track of the words within the block, which have been modified by the thread accessing the cache. This approach is expensive and complicates the hardware significantly. The compiler based approach is significantly simpler.

9.4.7 Atomic Operations

Till now, we have been using the coherence protocol to implement the cache coherence axioms. However, let us now use it to implement advanced functionalities that most parallel applications require. Let us consider a realistic scenario, where we are running a multithreaded banking application. The code for updating the balance in an account will be similar to that shown in Listing 9.1. We shall prove that this code is erroneous when run in a multithreaded setting, and we need *atomic operations* to ensure that this code works correctly. Let us devote this section to the study of such operations.

Listing 9.1: Code to update the balance in an account

```
void update (int amount) {
    balance += amount;
}
```

This code for the *update* function looks simple; however it is not safe in a multithreaded environment. To understand the reasons for this, let us look at an expanded version of the same code, where each line corresponds to a statement in assembly. Here, all the variables starting with a 't' stand for temporary variables that are assigned to registers.

```
1 void update (int amount) {
2     t1 = balance;      /* load instruction */
3     t2 = t1 + amount; /* add instruction */
4     balance = t2;      /* store instruction */
5 }
```

We replaced one C statement with three instructions: one load, one add, and one store. Let us now see what will happen when two copies of the same code run on two different threads. The execution is shown in Figure 9.49. For the ease of understanding, in thread T_2, we use a different set of temporary variables: $t3$ and $t4$. The instructions are numbered *1*, *2*, and *3* respectively for thread T_1, and *1'*, *2'* and *3'* for thread T_2.

If the set of instructions in T_1 run before or after the set of instructions in thread T_2, then there is no problem. However, in Figure 9.49, both the threads are trying to credit a value of ₹100 to the account[2]. We have assumed that the starting balance is ₹0. In this case, the final balance should be ₹200 irrespective of the order in which the threads credit the amount. However, because of the overlap in the execution of the functions, it is possible that both the threads read the value of the *balance* variable to

[2]₹is the symbol for the Indian rupee

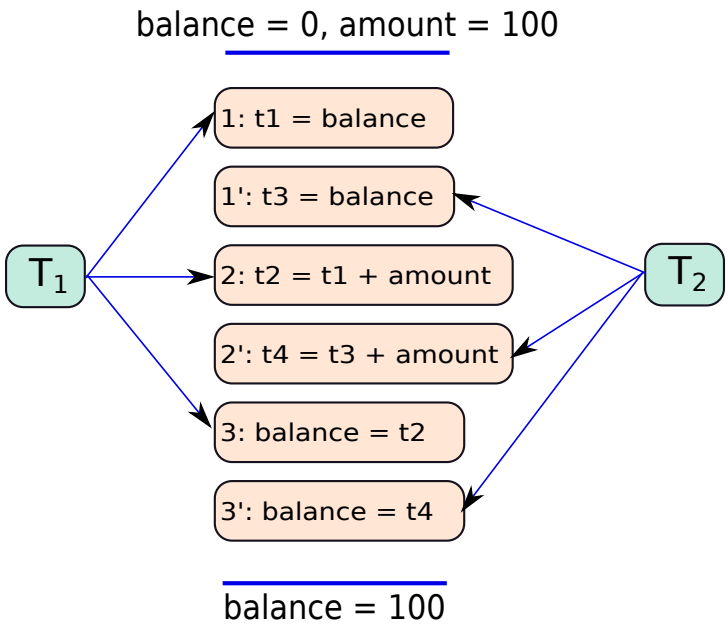

Figure 9.49: Two threads executing the code to update the balance at the same time

be 0. As a result, $t1$ and $t3$ are 0. Subsequently, $t2$ and $t4$ are set to 100, and the final balance is set to 100. This is clearly the wrong answer and this is happening because we are allowing an overlap between the executions of the *update* function in both the threads. There is a need to *in some way* lock the set of instructions such that we do not allow the same set of instructions to be executed concurrently by another thread. No two threads should be executing the instructions in the *update* function concurrently. We need a mechanism to ensure this.

A piece of code that does not allow two threads to execute it concurrently is known as a *critical section*. In this case, we need to create a critical section and insert these three statements in it such that only one thread can execute them at one time. Almost all languages today that support parallel programming also support the notion of critical sections. Without supporting critical sections it is not possible to write most parallel programs.

Definition 79
A critical section *is a region of code that contains contiguous statements, and all the statements in the critical section execute atomically. After thread t starts executing the first instruction in a critical section, it is not possible for another thread to execute an instruction in the same critical section till t finishes executing all the instructions in the critical section.*

Lock and Unlock Functions

Most critical sections use the lock-unlock paradigm. Here the idea is that we *lock* a memory address before entering the critical section. Locking a memory address is often tantamount to setting its value to 1 from 0. Then, when a thread leaves the critical section, it needs to unlock the memory address, which means setting its value back to 0 from 1. Let us call this memory address as the *lock address*.

However, if the lock is acquired (value set to 1), which means that there is already another thread

executing the critical section, then in that case, we need to wait till that thread has left the critical section and released the lock: performed an unlock operation. The thread that is trying to acquire the lock keeps trying to acquire the lock till it is free. Let us represent this situation pictorially in Figure 9.50. Here, we observe a call to the lock and unlock functions before we enter and exit the critical section respectively.

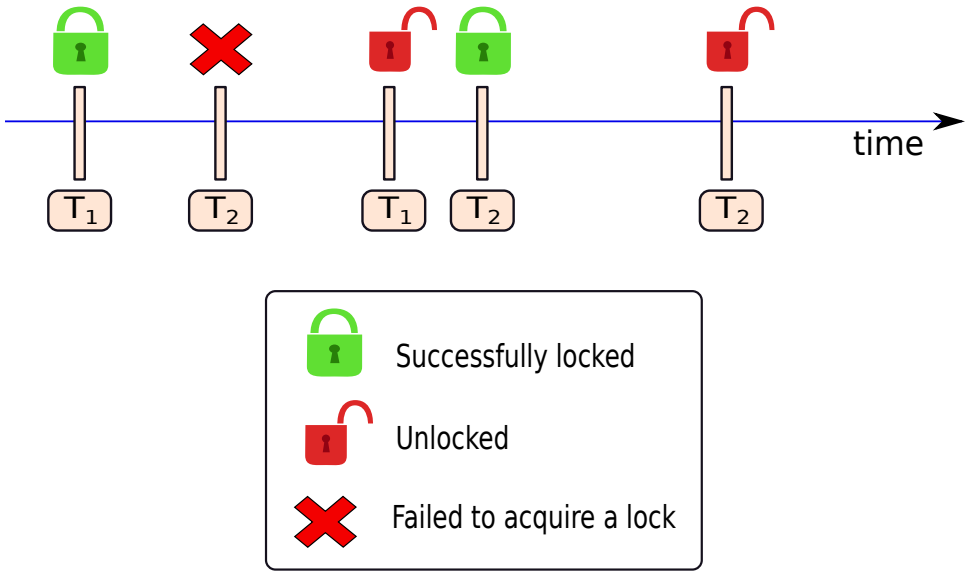

Figure 9.50: A timeline that shows two threads acquiring and releasing a lock

Before discussing the details and corner cases, let us look at the implementation of the lock and unlock functions. Let us make a simplistic assumption that there is one lock address in the entire system, and the associated lock needs to be acquired before we enter the critical section. The assembly code for the lock and unlock functions is as follows. Let us assume that the address of the lock is in the register $r0$. The *bne* instruction means branch-if-not-equal.

```
1  .lock:
2      mov  r1, 1
3      xchg r1, 0[r0]
4      cmp  r1, 0
5      bne  .lock
6      ret
7
8  .unlock:
9      mov r1, 0
10     xchg  r1, 0[r0]
11     ret
```

The key instruction in the lock function is the atomic exchange instruction called *xchg*. Note that till now we have not introduced this instruction. The atomic exchange instruction atomically exchanges the contents of a register and a memory location. The keyword here is *atomic*. This operation appears to happen to other threads instantaneously. No thread can interrupt the operation in the middle or observe any intermediate state. Let us now explain how to use this instruction to realize a lock function.

We first set $r1$ to 1, and then atomically exchange the contents of $r1$ with the lock variable (address stored in $r0$). If the lock is free, which means that no thread has currently acquired it, then the contents of the lock variable will be 0. After the exchange, the lock variable will contain 1, because we are exchanging its contents with register $r1$'s contents, which was set to 1 in Line 2. The interesting thing is that after the exchange operation, $r1$ will contain the earlier value of the lock variable. We compare it with 0 in Line 4. If the comparison is successful, which means that the lock variable contained 0, then it means that the current thread changed its value from 0 to 1. It has thus acquired the lock by successfully changing its status. In the other case, when the comparison fails, it means that the lock was already acquired – the value was already equal to 1. If we find that we have not acquired the lock, then there is a need to try this process again, and thus we loop back to the beginning of the lock function. If we have successfully acquired the lock function, we return to the caller function and start executing the critical section.

The unlock function is comparatively simpler. In this case, we just need to release the lock. This is as simple as setting the value of the lock variable to 0. Other threads can then acquire the lock by setting its value to 1. The key point here is that instead of using the regular store instruction, we use the atomic exchange instruction that also contains a fence. The idea here is that the fence ensures that when other threads see the unlock, they will also see all the reads and writes that have happened within the critical section, regardless of the memory model.

First, let us convince ourselves that this mechanism genuinely ensures that only one thread can execute the critical section at a given point in time. This property is also known as *mutual exclusion*. Let us try to formally prove this.

Theorem 9.4.7.1 *The algorithm with the lock and unlock functions ensures mutual exclusion.*

Proof: Assume that two threads T_1 and T_2 are in the critical section at the same time. With no loss of generality, let us assume that T_1 got the lock first and then T_2 got it. This means that T_1 set the value of the lock address to 1 from 0. When T_2 executed the atomic exchange instruction, it must have seen the value of the lock to be 1, which was set by T_1. There is thus no way that it could have seen the lock's value to be 0, because T_1 is still there in the critical section. If T_2 had seen the value of the lock to be 1, it could not have entered the critical section. Hence, the hypothesis is wrong, and we thus have a proof by contradiction. ∎

Implementing Atomic Operations

Let us now look at implementing atomic operations like atomic exchange in cache coherent systems. Irrespective of the atomic operation, the method to implement it is roughly the same. The first step is to get write access to the block that contains the lock variable. In an invalidate protocol with a directory, which is the standard, the processing begins with a write miss received at the directory. It invalidates all the copies of the block, and after collecting acknowledgements from all the sharers it sends a message to the requesting cache. Once the requesting cache receives a go-ahead from the directory, it sets the state of the line to M (modified).

Instead of doing a write, in this case we proceed with the atomic operation. Most atomic operations are read-modify-write operations. We read the value of a memory location, compute the new value, and write it to the memory location. We proceed with all the steps once the line's state is set to M. The ideal case is where all the steps complete without interruption and the atomic operation completes.

The worst case is when in the middle of the execution of the atomic operation another cache sends a read miss or write miss. We clearly cannot abandon an atomic operation in the middle. There are several strategies to deal with this. The simplest strategy is that the cache and the core executing the atomic

operation hold off sending the acknowledgement to the directory till the atomic operation completes. Another idea is a lease based approach. The directory assumes that a cache requesting for a block in the M state should at least get κ cycles to work on the contents of the block. Meanwhile, if the directory receives any other request, it simply queues it. If κ is enough for an atomic operation to execute, then we need not rely on acknowledgements.

Atomic operations are often synchronization operations (see Section 9.3.5). This means that they also act as fence instructions. This is required because such operations are typically used to implement critical sections or implement other important parallel programming primitives: this requires them to behave like a fence and enforce some memory orders for the instructions before and after them in program order. This aspect of their execution further increases their overhead.

Efficient Spin Locks

Our simple lock-unlock algorithm does indeed guarantee mutual exclusion. However, it is not a very efficient algorithm because it keeps on trying to acquire the lock in a loop – such a pattern is known as a *spin lock*. Let us look at some flaws of such naively implemented spin locks.

1. Each attempt requires the thread to perform memory, arithmetic, and synchronization operations repeatedly. This consumes a lot of power and is slow.

2. The other problem with a spin lock is that threads basically do useless work when they are waiting for a lock. Even though the processor might perceive them to be busy; however, they are actually not doing any useful work. Most processors will not be able to detect this pattern, and thus will not schedule instructions from other threads. In modern locks used by the Linux operating system, the code of the lock is written in such a way that after a certain number of iterations, the thread notifies the OS that it is ready to sleep. The OS can then schedule another thread or another process on the core.

3. There is a possibility of *starvation*, which means that a thread might never be able to acquire a lock. It might always lose the competition to another thread. Modern locking algorithms have a notion of fairness, where they ensure that a thread does not have to wait forever. However, they are far more complex as compared to the simple code that we have shown. Interested readers can take a look at the book by Herlihy and Shavit [Herlihy and Shavit, 2012] for a discussion on modern algorithms to implement locks.

Creating a fair locking algorithm is out of the scope of this book. This requires a complex locking algorithm, where we maintain an order between the requests, or ensure that the system somehow increases the priority of threads that have been waiting to get a lock for a long time. Let us instead focus on the time and power overheads.

In Linux, locks typically wait for a fixed duration, typically 100 μs, and then automatically send an interrupt to the OS kernel. The OS kernel puts the thread to sleep and schedules some other thread. This ensures that threads waiting for a lock do not unnecessarily tie up a core. Furthermore, this reduces the power overheads of spin locks significantly. The sad part is that this also makes our parallel programs slower. Let's say we have 10 threads, and we want all of them to finish a critical section, before we can make progress. If one of the threads gets swapped out of the core by the OS, then it will not be able to execute even if the lock becomes free. We need to wait for the OS to reschedule the swapped thread. This will unnecessarily block the entire set of 10 threads. Let us thus slightly speed up the execution of the basic lock primitive.

The main problem with a basic spin lock that uses the exchange instruction is that in every iteration, we try to set the value of the lock variable using an expensive synchronization instruction. This means that we need to send a write miss message on the bus, and wait till we get the data in the M state. Recall that in the M state, a cache owns the block exclusively, and it can modify its contents. The main problem with modern write-invalidate protocols is that their performance dips if multiple threads are

desirous of writing to a block simultaneously. Because of exclusive ownership in the M state, the block keeps bouncing between caches, and this causes a lot of network traffic as well as slowdown. This can be reduced by creating an optimized version of a spin lock.

Let us create an algorithm that tries to write to a block only if it feels that there is a high probability of the atomic exchange operation being successful, which alternatively means that there is a high probability of lock acquisition. To achieve this, let us first test if the value of the lock variable is 0 or not, and only if it is 0, let us make an attempt to acquire the lock. This will drastically reduce the number of invalidate messages and the number of times we need to use synchronization instructions such as atomic exchange. The code to implement this concept is as follows.

```
1  /* the address of the lock is in r0 */
2  .lock:
3      mov  r1, 1
4
5  .test
6      /* test if the lock is 0 */
7      ld   r2, 0[r0]
8      cmp  r2, 0
9      bne  .test
10
11     /* attempt an exchange only if the lock is free */
12     xchg r1, 0[r0]
13     cmp  r1, 0
14     bne  .test
15     ret
16
17 .unlock:
18     mov r1, 0
19     xchg  r1, 0[r0]
20     ret
```

In this case, we have added three extra lines: Lines 7 till 9. The aim of these lines is to first read the value of the lock variable, check if it is equal to 0, and then exit the loop if the value of the lock variable is found to be equal to 0. Assume that another thread has acquired the lock. Then it will have the lock variable in the M state. The first time that the current thread reads it, the blocks in both the caches will transition to the S state. This requires a read miss message. However, after that the current thread will keep on reading the block, and since it is in the S state, this will not require any messages to be sent to the directory nor do we need to use the atomic exchange instruction to test if the lock is free or not. This is far more power efficient and also the messages on the NoC will reduce significantly. This method is called test-and-exchange (TAX). Once, we read the value of the lock variable as 0, we are sure that the thread that was holding the lock has released it.

We can then proceed to Line 12, where we try to perform the atomic exchange. Here, if we are successful, then we are deemed to have acquired the lock. Note that it is possible that two threads may have realized that the lock is free, and both of them may try to execute the atomic exchange operation in Line 12 concurrently. In this case, only one thread will be successful. The other thread needs to start the entire operation of trying to acquire the lock once again.

It is true that this algorithm increases the time it requires to acquire a lock if there is no contention. This is because of the additional test step. However, in the case of a contended scenario, we will need to execute multiple exchange instructions using the basic algorithms that we have proposed. This is slow because of the inherent fence operation and will cause many write misses. With the TAX mechanism, we have replaced write misses with read hits because till a thread owns the lock, the rest of the threads will continuously read the lock variable, and find it to be in their caches in the S state: no messages are

sent to the directory. A read hit is power efficient and does not lead to NoC traffic. Once we have some hope of getting the lock, we issue the expensive atomic exchange instruction.

Definition 80
A spin lock is a locking algorithm where we repeatedly check the value of a lock variable stored in memory, in a loop. The advantage of a spin lock is that the threads get to know very quickly when a lock is released. However, the disadvantage is that a thread keeps on executing the same code over and over again in a loop without doing any other useful work. This wastes power. Furthermore, the CPU and system software also falsely believe that a thread is doing useful work and thus do not schedule other threads on the same core.

Other Atomic Operations

We have seen the atomic exchange operation, and we have also seen how we can implement a lock with it. This is not the only kind of atomic operation. There are many more types of operations that can be used for numerous kinds of operations. Implementing a lock is one of the simplest operations in the field of parallel and concurrent algorithms. There are far more complicated algorithms (refer to [Herlihy and Shavit, 2012]) that require more complicated atomic operations. Let us list some such atomic operations with a snippet of pseudo code describing their operation in Figure 9.51.

Figure 9.51 shows a set of atomic operations starting from the simple test-and-set operation to the elaborate LL/SC operation. Here, LL stands for load-linked and SC stands for store-conditional. In an LL-SC pair, each of these operations are atomic operations. They work as follows. An LL operation loads a value from memory like a regular load operation. However, in addition to this, it also sets a flag in the cache line containing the local copy of the block indicating that an LL operation has been executed. Subsequently, we can execute other instructions depending upon the logic of the program. It is necessary to execute an SC operation at a later point in time. This operation is mostly similar to a regular store operation with one critical difference. The operation returns a value and is conditional. If the block has not been modified after the last LL operation, then the paired SC operation returns 1, and also completes its operation. However, if the block has subsequently been modified by the same thread or another thread then SC returns 0 and does not execute the store operation. We can easily implement a lock using the LL/SC operations as shown[3] in Example 12.

We can implement all the operations by slightly modifying the cache coherence protocol such that we finish the execution of the atomic operation by acquiring the lock variable in the M state. For LL/SC we need to set a bit in the cache line that contains the lock variable. Whenever a write miss arrives from another cache, we set this bit to 0, otherwise this bit remains 1. Now, when we subsequently perform the SC operation, we check the bit in addition to getting write access to the block. If the bit is still set to 1, the SC operation is successful. If the bit is 0, or if the block has been evicted, then the SC operation fails: this means that there may have been an intervening write by another thread. If the SC operation is successful, the write operation to update the lock variable is effected. Both LL and SC operate atomically.

Example 12 *Implement the lock and unlock functions using the LL/SC primitive and fence instructions. Assume that the address of the lock is stored in register r0. Reduce the number of NoC messages by first testing the value of the lock.*

Answer:

[3]The code for acquiring and releasing a lock for obvious reasons cannot use locks.

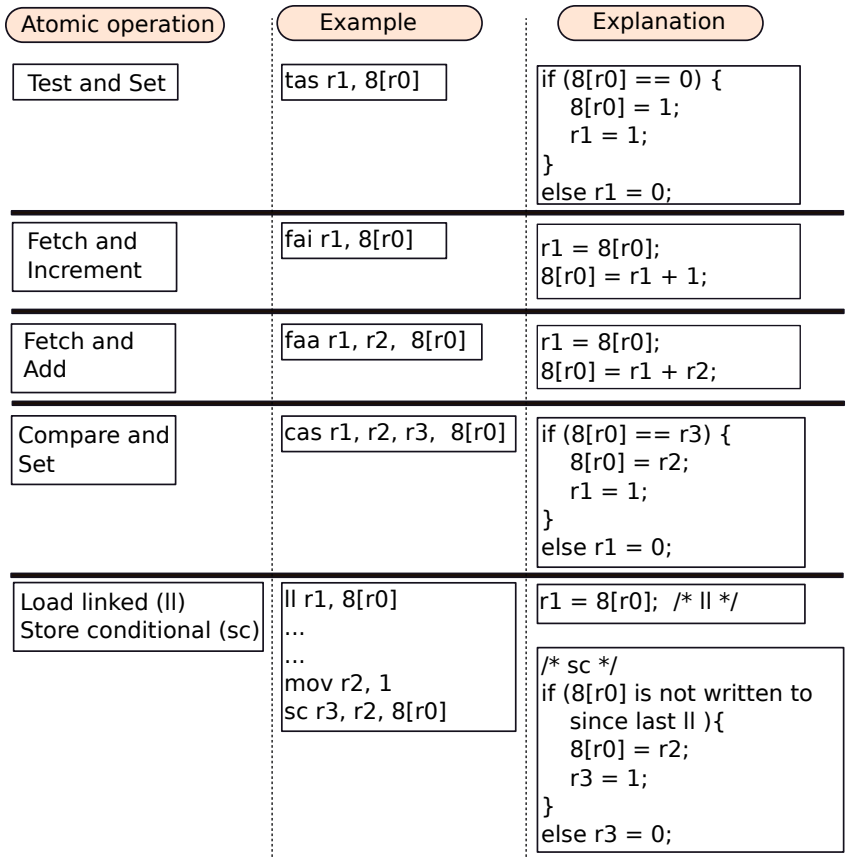

Figure 9.51: Different types of atomic operations

lock function

```
.lock:
    ll  r1, 0[r0]
    cmp r1, 0
    bne .lock     /* If the lock is not free
                     iterate once again */

    mov r2, 1
    sc  r3, r2, 0[r0]
    cmp r3, 1
    bne .lock     /* iterate if sc is not
                     successful */
    ret
```

unlock function

```
.unlock:
    mov r1, 0
    fence
    st  r1, 0[r0]
```

```
    ret
```

On similar lines, we can use other atomic operations to implement locks. Before the reader asks, "Why do we have so many types of atomic operations?", let us answer this question. The trivial answer that comes to the mind is that some operations do some computations with fewer lines of code. For example, we can always implement a fetch-and-add operation with a compare-and-set (CAS) operation. However, this will be cumbersome. As a result, having more instructions will allow us to write simple and elegant code. This is however just the superficial part of the story. There is a much deeper answer, which is that different atomic operations have different amounts of power. This means that some operations are less powerful and some other operations are more powerful. We can always implement a less powerful operation with a more powerful operation; however, we cannot do the reverse.

For example, operations such as test-and-set and atomic exchange are regarded as the least powerful. In comparison, compare-and-set and LL/SC are the most powerful. There is a spectrum of atomic operations whose power lies between them. Let us elaborate.

9.4.8 Lock-free Algorithms using Atomic Operations

Consider the problem of updating the bank balance once again. The crux of our argument was that if we use multiple RISC instructions to update the balance, then it is possible that due to conflicting operations by different threads, the final result can be wrong. Hence, we decided to wrap the code in a critical section such that only one thread can access it. Each critical section begins with a call to a lock function, where we set the value of the lock (memory address that contains it) to 1, and then it ends with a call to the unlock function. To ensure that we atomically update the value of the lock, we introduced atomic operations (see Figure 9.51).

Locks unfortunately have their share of problems. The biggest problem is that they do not allow *disjoint access parallelism*. This means that even if two threads need to update the balance of two separate accounts, only one thread can be in the critical section at any point in time. Note that if the accounts are different, it is possible for both the threads to execute their critical sections concurrently – this is not allowed. To solve this, we can have different lock addresses for different bank accounts. This solution will work for a simple update of a bank account. However, if we are implementing a parallel data structure such as a concurrent queue where multiple threads can enqueue and dequeue items concurrently, such approaches will not work. In general, using locks will ensure that only one operation can be done at any single time. Hence, using locks with a concurrent queue will effectively make it a sequential queue. Other disadvantages of locks include the possibility of starvation where a thread never gets the lock.

Additionally, with locks we can have deadlocks (no thread makes progress) as follows. Assume that there are two locks: A and B. Consider a situation where thread 1 holds lock A and tries to acquire lock B. Similarly, thread 2 holds lock B and tries to acquire lock A. In this case, none of the threads will be able to make progress. This situation is a deadlock. Furthermore, in most practical systems, instead of wasting power by continuously testing the value of a lock (spin locks), most operating systems put the thread to sleep. It takes a disproportionately long time to wake up the thread later.

It is possible to do better by using a set of algorithms known as non-blocking or lock-free algorithms that do not use locks at all. Most concurrent libraries are today written using such non-blocking algorithms. There is a general result that all operations on data structures such as stacks, queues, etc., can be implemented using lock-free algorithms (see [Herlihy and Shavit, 2012]. Note that even though lock-free algorithms are very promising, they are very hard to code and debug.

Let us show a simple lock-free algorithm for updating the bank balance with the compare-and-set (CAS) primitive. The format of the special CAS instruction is as follows: $CAS\langle reg4\rangle, \langle reg3\rangle, \langle reg2\rangle, \langle mem\rangle$.

If the contents of the memory location are equal to the value of $reg2$, then the value stored in the memory address is atomically set to the value of $reg3$. If the CAS is successful, then we set the value of $reg4$ to 1, else we set it to 0.

```
/*  address of balance is in r0
    the additional amount is in r1 */

.start:
    ld  r2, 0[r0]       /* r2 contains the balance */
    add r3, r2, r1      /* r3 contains the final balance*/
    CAS r4, r3, r2, 0[r0]  /* if (r2 == 0[r0]) 0[r0] = r3*/

    cmp r4, 0           /* test the result */
    beq .start          /* retry if the CAS fails */
```

In this case, we do not use locks. We repeatedly invoke the CAS instruction to set the value of the variable *balance*. If the CAS fails, then it means that some other thread has succeeded in updating *balance*, and we try again. In this case starvation is possible; however, this implementation is more efficient. If the lock is free, we need 5 instructions to finish the operation using the lock-free algorithm. Whereas in the implementation using locks, we require 5 instructions for the lock, 3 instructions for the unlock, 3 instructions for updating the balance, and 2 function calls.

Wait-free Algorithms

Lock-free algorithms are typically much faster than their lock based counterparts for implementing concurrent data structures. However, they have the problem of starvation. We can use *wait-free algorithms* that additionally guarantee that every operation completes in finite time. Wait-free algorithms are more complicated than their lock-free counterparts, and on an average are slower.

They work on the principle of helping. If any thread is not able to complete its operations, then other threads help it complete its operation. This ensures that there is no starvation.

Definition 81

- *A* lock-free *algorithm does not use locks. With such algorithms we can have starvation where a given thread may never complete its operation because other threads successfully complete their operations.*
- *A* wait-free *algorithm provides more guarantees. It guarantees that a given thread will complete its operation within a finite or bounded number of internal steps.*

Consensus Numbers and the Power of Atomic Operations

Definition 82 *The* consensus *problem is as follows. Let us assume that we have n threads. Each thread proposes a value. Ultimately all the threads choose a value that is one among the set of proposed values.*

The consensus problem is a very basic problem in concurrent systems. Its definition for an n-thread system is as follows. Let each thread propose a value. Eventually, all the threads need to agree on a single value that is one among the proposed values. It can be shown that a lot of real world problems are basically different variants of consensus problems. In fact, the heart of modern cryptocurrencies such as Bitcoin is a consensus problem. The basic problem that most transaction processing systems such as online payments solve is a consensus problem. As a result, solving the consensus problem is of paramount importance in concurrent systems, and moreover, it can be shown that many problems of interest can be mapped to equivalent consensus problems. The power of different atomic operations is based on who can efficiently solve the consensus problem in finite time in different settings.

This is quantified by the *consensus number* of an atomic operation, which is defined as follows. It is the maximum number of threads for which we can solve the consensus problem using a wait-free algorithm that uses the atomic operation and simple read/write operations. If the consensus number is k for a given atomic operation, then it means that it is theoretically not possible to write a wait-free algorithm that solves the consensus problem in a $k+1$ thread system.

Let us look at the consensus numbers of some common atomic operations.

Type of operation	Consensus number
Atomic exchange	2
Test and set	2
Fetch and add	2
CAS (compare and set)	∞
LL/SC	∞

From the definition of consensus numbers, it is clear that an operation with a lower consensus number cannot be used to implement an operation with a higher consensus number. This automatically implies that we cannot use test-and-set to implement CAS using a wait-free algorithm. The most powerful operations are CAS and LL/SC.

9.5 Memory Models

Let us quickly summarize what we have learned in the preceding sections with respect to memory models.

Way Point 12

- *There are four kinds of relationships between regular memory operations: ws, fr, po, and rf.*
- *ws and fr orders are global in most systems today because of the requirements imposed by PLSC.*
- *Different processors relax different orders within po and rf. They thus have different memory models. If a given order is not global, it is said to be* relaxed.

Given that we can relax different orders that are a part of rf and *po*, we can create a variety of memory models. Different models have different trade-offs between flexibility and performance. Let us look at each of these relaxations from an architectural perspective.

9.5.1 Relationships in *rf*

Let us look at cases where we need to relax the rfi and rfe relationships. Recall that the rfi relationship is between a write and a successive read to the same address in the same thread, whereas the rfe

relationship is between a write and a read to the same address across threads. The rfe edge will be global if we have atomic writes because the write will appear to happen instantaneously and thus all the threads will agree on the write-to-read order.

When we use a write buffer (see Section 7.1.7), we are breaking the rfi order. Consider the situation, where we have a write and a subsequent read from the same address (in the same thread). The write operation is not made visible to the rest of the cores immediately. The write is sent to the write buffer and is not immediately broadcast to the rest of the cores. However, a later read operation can read its value and make progress. This effectively means that the read is visible globally, before its earlier write. The earlier write operation is visible to the rest of the cores, when it is ejected from the write buffer. From the point of view of the rest of the cores, the write executes after the read. Hence, the rfi relation in this case is not global. We have a similar case when we have forwarding in the LSQ. A later read gets the value from the LSQ and moves ahead, whereas the write needs to wait till the instruction gets committed.

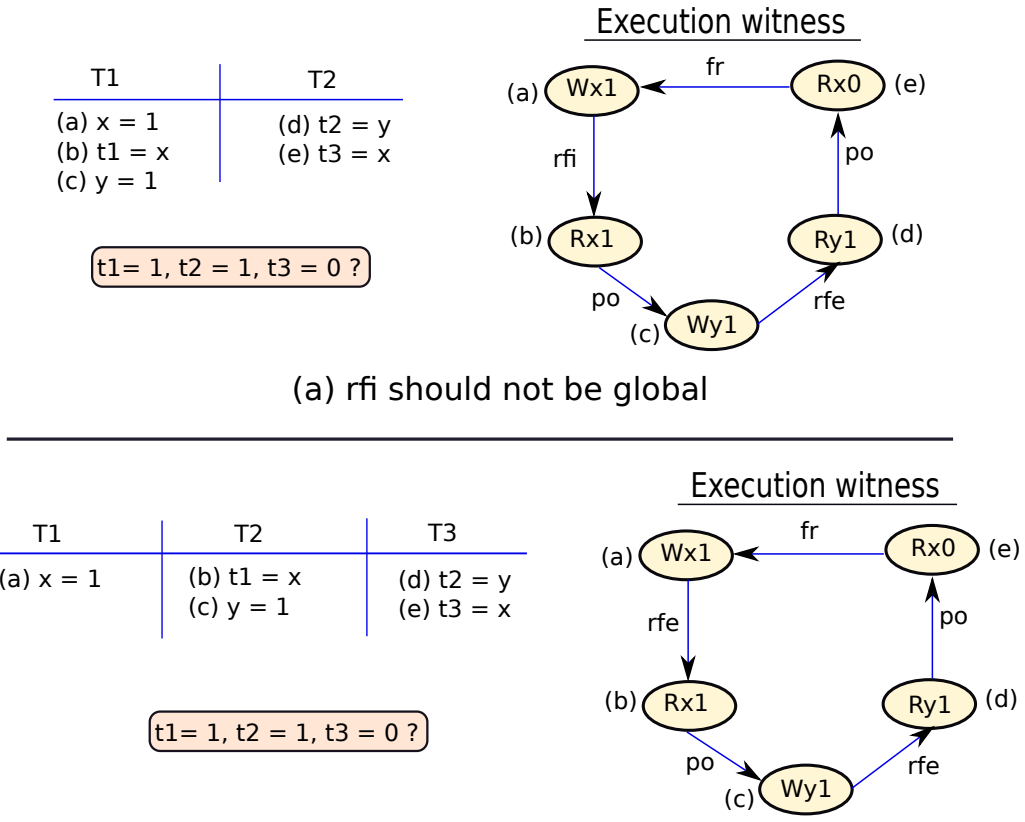

Figure 9.52: Execution witnesses: rfi and rfe orders

Consider the execution witness as shown in Figure 9.52(a). In this case, the read and write are a part of the same thread. Let us assume that the rfi relation is global. Additionally, the rfe edge, which is a write to read edge across threads is also assumed to be global because we are assuming atomic writes in this example. Now, assume that we do not have a program order edge between a write and a read. In We can thus add only an rfi edge between $Wx1$ and $Rx1$. Then we add a po edge between $Rx1$ and $Wy1$ because in this case we assume that a read to write program order is global. We then add an

rfe edge between $Wy1$ and $Ry1$. Finally, we add a program order edge (po_{RR}) between $Ry1$ and $Rx0$, and then an fr edge between $Rx0$ and $Wx1$. We add an fr edge because the instruction $Rx0$ reads an earlier value of x. Now, we see that we have a cycle in the execution witness. Since many processors obey the RR and RW program orders, fr is global, and we have assumed rfe to also be global because of atomic writes, the only relation that we can relax is rfi for this execution to be valid.

In almost all OOO processors with atomic writes, this execution will be valid because the rfi edge is not respected. We say that an order is *respected* if it holds globally. In fact, whenever we delay earlier writes and use structures like write buffers, rfi is not global: this execution will be valid.

Now consider the example in Figure 9.52(b). Assume that the RW and RR program orders hold. Since the fr edge is global, the only edge in the graph that can be relaxed is the rfe edge. To avoid a cycle, the rfe order needs to be relaxed. This means that this execution is valid in a system with non-atomic writes. If we have atomic writes, this execution is not allowed.

9.5.2 Write-to-Read Program Order

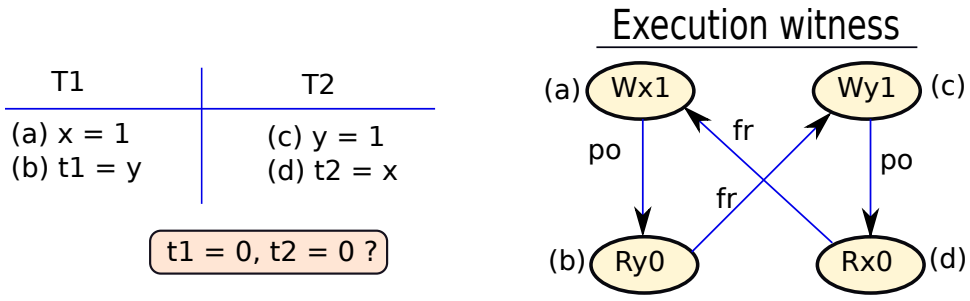

Figure 9.53: Execution witness: po_{WR} should not be global

In most conventional OOO pipelines, we send stores to the memory system once they reach the head of the ROB and are ready to be committed. We, however, do not stall later loads to different addresses. If there is no chance of a forwarding in the LSQ, the loads are sent to the memory system. This means that later loads can overtake earlier stores. In other words the $W \rightarrow R$ program order is not respected. In fact, the key aim of having an LSQ is to allow later loads to overtake earlier stores. Hence, in almost all practical memory models this ordering is relaxed. Figure 9.53 shows an example along with its execution witness, where the $W \rightarrow R$ program order edge needs to be relaxed for the execution to be valid. Readers are welcome to run this code on any multicore machine. We claim that they will see the output $((t_1, t_2) = (0, 0))$ at least once.

9.5.3 Write-to-Write Program Order

Let us now consider the write-to-write ($W \rightarrow W$) program order. If we have writes to different addresses, there is no requirement for them to take effect in program order for a single-threaded program. The correctness of the thread is not dependent on the order in which these writes are executed. However, for the rest of the cores, the order matters.

Such an order can break for a variety of reasons. It is true that from the point of view of the core, we commit write instructions in program order. However, it does not mean that they are sent to the memory system in that order. Consider the write buffer. Let's say we have two writes, W_1 and W_2, in a thread where W_1 is before W_2 in program order. If there is a write buffer entry for the address of W_1, then the write will be written to the write buffer entry. However, if the entry for W_2 is not there in the write buffer, we have an option of sending it directly to memory, instead of freeing a write buffer entry. In this case, W_2 appears to happen before W_1 to other cores, which is not true.

The $W_1 \overset{po}{\rightarrow} W_2$ relation can also break because of messages in the NoC. It is possible that the write messages might get reordered. Thus, the $W \rightarrow W$ order will not remain global.

A guaranteed way to ensure that the write to write order is maintained is to make use of acknowledgement messages. The assumption is that an acknowledgement message is sent to a core after the write becomes globally visible. This means that the write is visible to all the threads. In the case of $W_1 \overset{po}{\rightarrow} W_2$, we wait for the acknowledgement of W_1 and then send the write W_2 to memory. The main problem with such acknowledgements is that they make a write more expensive. The write unnecessarily blocks the pipeline till its acknowledgement arrives. This delays later instructions ultimately reducing the IPC.

There are other mechanisms as well that ensure that the memory system is designed in such a way that later writes do not overtake earlier writes. These require changes to the write buffers, MSHRs, and the NoC. This is why most weak memory models in use today do not ensure the $W \rightarrow W$ order. The Intel TSO model is an exception to this rule. It obeys the $W \rightarrow W$ ordering.

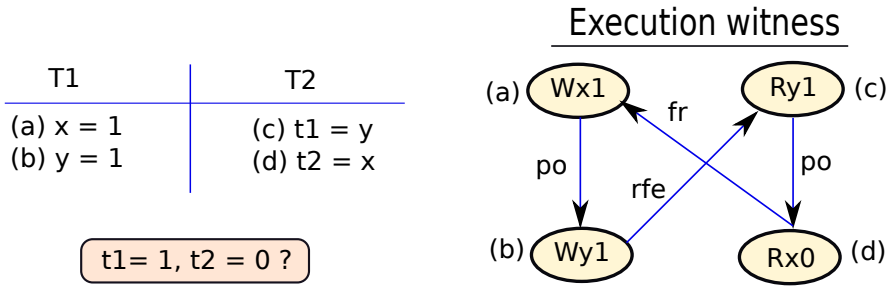

Figure 9.54: Execution witness: po_{WW} edges should not be global

Consider the example in Figure 9.54. The execution witness has a cycle. If we assume that rfe is global (writes are atomic), and fr is also global, then we have two more edges left: po_{WW} and po_{RR}. Assume that po_{RR} holds. Then the only edge that we can relax is the po_{WW} edge between $Wx1$ and $Wy1$ to make this execution valid. In most modern processors that have a weak memory model, such $W \rightarrow W$ program orders are not global. Hence, this execution is valid. However, in processors that follow the *total store order* memory model (mostly Intel processors), where the order of writes (stores) is global, this execution is not allowed.

9.5.4 Read-to-Read Program Order

The read-to-read ($R \rightarrow R$) order is also not respected by many OOO processors as of 2020. To ensure that this ordering holds we need to issue load operations in program order. Note that this decreases performance because it is possible that the address of a later load might be computed before the address of an earlier load. In this case, it is not fair from a performance point of view to make the later load wait for the earlier load. Recall that we are considering loads to different addresses.

Even though ensuring this order is easy, which is by making modifications to the LSQ, we should realize that we would be sacrificing some amount of performance. This is not acceptable in high performance processors.

Let us understand this order with an example (refer to Figure 9.55). We have three threads running on a machine with atomic writes (rfe is global). Assume that the po_{RW} edge is global, which is hard to relax even in OOO processors (discussed in the next subsection). Given that the fr edge is always global, the only edge that we can relax is the po_{RR} edge. On a machine with weak ordering that does not respect the $read \rightarrow read$ program order, this execution would be valid. However, if the LSQ ensures a strict ordering between reads, this execution is not possible.

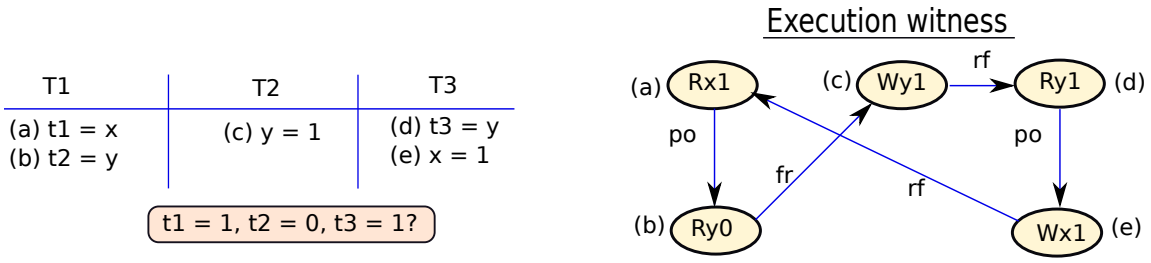

Figure 9.55: Execution witness: po_{RR} edge should not be global

In an OOO processor, we ideally do not want to stall later reads because of earlier unresolved reads (address is not computed). We would ideally like to send read operations to the memory system as soon as possible. This is because many instructions are typically dependent on the result of a read operation. Hence, the program order between read operations to different addresses is seldom respected.

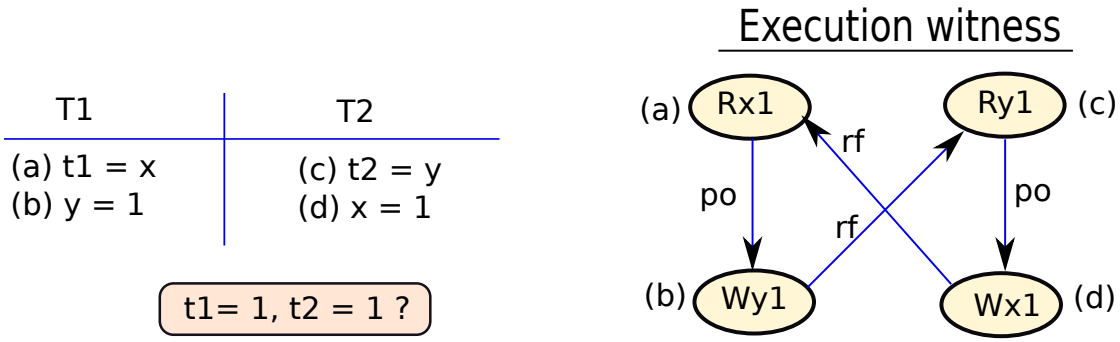

Figure 9.56: Execution witness: po_{RW} edge should not be global

9.5.5 Read-to-Write Program Order

In most OOO processors, this order is maintained; however, there are exceptions. In a conventional OOO processor the earlier read needs to have read its value and left the pipeline before the later write can write its value at commit time. To reverse the order, we need to think of a situation where the read operation appears to read its value after the write. This can happen in systems where writes are sent to the memory system early. We can design systems where an instruction can go ahead and write to an entry in the write buffer after we know that it is on the correct branch path. In this case, writes can be visible before earlier reads have read their value. Another case is when we use load-value prediction, where the prediction is validated after the load has committed. Such a situation is shown in Figure 9.56, where we assume atomic writes (rfe is global). The only edge that we can relax to avoid a cycle is the po_{RW} edge. Now that we have understood the implications of relaxing different orderings, let us look at the orderings that are relaxed by different memory models.

9.5.6 The Special Case of *rfi* in SC

Consider a sequentially consistent machine. What will happen if the rfi edge is not global? Will it still remain sequentially consistent? This will allow us to use LSQ forwarding in an SC machine. A core or thread can effectively read its own write *early*.

Consider a write operation W and a later read operation R that accesses the same address. They belong to the same thread. Let there be intervening operations of the form $O_1 \dots O_n$. Furthermore, let us assume that none of $O_1 \dots O_n$ are read operations that access the same address, and there are no rfi dependences between them (we can always find such a pair if W and R access the same address). Then we can write $W \overset{ghb}{\rightarrow} O_1 \overset{ghb}{\rightarrow} O_2 \dots O_n \overset{ghb}{\rightarrow} R$ because program order holds in SC for the cases that we consider. This means that $W \overset{ghb}{\rightarrow} R$. In this case, it does not matter if rfi is global or not because they are globally ordered anyway. Hence, rfi being global only matters when W and R are consecutive operations – there are no intervening operations. Note that in this case, we do not add a *po* or rfi edge between them. The question is whether the execution will still be in SC?

SC will be violated only when other threads see the read by R before seeing the previous write by W. Note that this problem will not happen in the same thread because as far as future operations in the thread are concerned – R is taking effect in program order. Now, consider the first operation O in another thread, which is reachable from R in the execution witness. It is either reachable via an $R \overset{fr}{\rightarrow} O$ edge or from an edge from another instruction O' in the same thread where we have $R \overset{po}{\rightarrow} O' \overset{ghb}{\rightarrow} O$. We use the $\overset{ghb}{\rightarrow}$ edge here because the nature of the edge between O' an O does not matter. Now, consider the first case. We need to also have $W \overset{ws}{\rightarrow} O$ (by the definition of fr). In the second case, we will have $W \overset{po}{\rightarrow} O' \overset{ghb}{\rightarrow} O$. In both cases, we will have $W \overset{ghb}{\rightarrow} O$. Hence, as far as O is concerned, both W and R happen before it, and R appears to have executed after W because it returns the value written by it. The fact that we relaxed the rfi edge between consecutive instructions is not visible to the same thread or to other threads. Hence, the execution still is in SC because the rest of the conditions for SC hold.

We can thus conclude that in SC, a thread can read its own writes *early*.

Model	Program order (*po*)				rf	
	po_{WR}	po_{WW}	po_{RR}	po_{RW}	rfe	rfi
SC						✓
TSO	✓					✓
PC	✓				✓	✓
PSO	✓	✓				✓
Weak const.	✓	✓	✓	✓		✓
RC	✓	✓	✓	✓		✓
PowerPC	✓	✓	✓	✓	✓	✓
ARM	✓	✓	✓	✓	✓	✓
A ✓ indicates that the ordering is relaxed						

SC	Sequential consistency	Weak const.	Weak consistency
TSO	Total store ordering	RC	Release consistency
PC	Processor consistency	PowerPC	PowerPC's memory model
PSO	Partial store ordering	ARM	ARM v7 memory model

Table 9.4: Popularly used memory models (adapted from [Adve and Gharachorloo, 1996])

9.5.7 Popular Memory Models

Table 9.4 lists some common memory models. Other than SC, each model relaxes some order in pursuit of better performance. Each memory model can be characterized by the orderings that it respects and the orderings that it relaxes. To find if a given execution is allowed by a memory model or not, we simply need to create an execution witness, add the edges that are a part of the memory model, and see if there is a cycle or not. If there are no cycles, then the execution is allowed by the memory model,

otherwise it is not allowed. For SC we need to add all the po, ws, rf, and fr edges, whereas for other models we add fewer edges. They thus allow more executions.

Note that memory models are not necessarily artifacts of a hardware design, they can be used to describe software systems as well. Consider the comments on a news story. It is non-intuitive to see replies to comments before seeing the comments themselves. This is an example of consistency in the software world. We have defined memory models from the point of view of threads. Hence, the underlying substrate does not matter – it can either be software or hardware.

Consider a system such as the Java virtual machine (JVM), which runs Java programs by dynamically translating Java byte code to machine code. It also needs to implement a memory model such that programmers know what orderings are preserved in the final execution. In fact a lot of compiler optimizations are dependent on the memory model. For the purposes of increasing efficiency, compilers routinely reorder instructions subject to the uniprocessor access constraints. This reordering can violate the program order relations of the memory model. Hence, the memory model interacts with compiler optimizations as well. As a thumb rule, readers should assume that any entity in the stack starting from the compiler to the virtual machine to the actual hardware can reorder instructions. Given the way that we have defined memory models, the point of view of the programmer and the final outcome of the program determine the memory model. Let us proceed with these assumptions in mind.

The gold standard of memory consistency models is sequential consistency (SC), which is mainly a theoretical model and is used to reason about the intuitive correctness of parallel programs and systems. Implementing SC is expensive in terms of performance, and thus is almost always impractical. Almost all optimizations are precluded in SC, and thus very few mainstream processors support SC. The only exception to this rule has been the MIPS R10000 processor that provided sequential consistency. We shall see in Section 9.6 that there are methods to give the programmer an illusion of sequential consistency even though the underlying hardware has a relaxed memory model.

As compared to SC, the second model, TSO (total store ordering), has seen more commercial applications. The Intel x86 and the Sun Sparc v8 memory models broadly resemble TSO [Alglave, 2012]. This model relaxes the po_{WR} and rfi relations. TSO can thus be supported by OOO processors, and we can seamlessly use LSQs and write buffers. Note that the rest of the program orders still hold and writes are atomic.

Many multiprocessor systems (particularly software systems) relax the TSO model to allow for non-atomic writes even though they do not relax the po_{WW} edge. This means that writes from the same thread are seen in program order, even though a thread can read the value of a write (issued by another thread) before all the threads see it – a thread can read another thread's write early. Implementing atomic writes is actually difficult in large systems where we can have numerous cached copies. Thus, it sometimes makes sense to relax the requirements of write atomicity. The Processor Consistency (PC) memory model falls in this class; it supports non-atomic writes.

The PSO (partial store ordering) model on the other hand supports atomic writes but relaxes the po_{WW} edge. It was supported by some Sun SPARC v8 and v9 machines. The advantage of relaxing the $write \rightarrow write$ order is that we can support non-blocking caches. A later write can be sent beyond the MSHR to the lower levels of the memory systems, while an earlier write waits at the MSHR. This optimization allows write operations to be reordered in the NoC as well. Note that read and write operations are fundamentally different. A read operation is synchronous, which means that the core gets to know when the value arrives. It is thus easy for it to enforce an order between a read instruction and any other instruction. However, writes are by nature asynchronous. Unless we have a system that sends write acknowledgements, a core has no idea when a write takes effect. Thus, enforcing an order between writes and other operations is difficult. Hence, PSO relaxes both the po_{WR} and po_{WW} orders. This simplifies the design of the memory system and the NoC.

The next model is called *weak consistency*, which is a generic model where all the orderings are relaxed other than write atomicity. Many RISC processors that are used to implement large multicore systems use some variant of weak ordering. Note that here write atomicity is the key; it is not compromised.

All the memory models that we have seen up till now define synchronization instructions, and all of

them respect the ordering between normal instructions and synch instructions. This means that they respect the following orders:

$$synch \overset{ghb}{\rightarrow} (read \mid write \mid synch)$$
$$(read \mid write \mid synch) \overset{ghb}{\rightarrow} synch$$

Let us now introduce another model called release consistency (RC) that was designed to implement critical sections efficiently. It supports the same orderings as weak consistency. However, it defines two additional synchronization operations – *acquire* and *release*. In other words, a synch operation can be an acquire, release, or any other synchronization operation. The orderings between these operations are as follows:

$$acquire \overset{ghb}{\rightarrow} (read \mid write \mid synch)$$
$$(read \mid write \mid synch) \overset{ghb}{\rightarrow} release$$

This means that we need to wait to complete an acquire operation, before any subsequent instruction can complete. This operation can be used for example to acquire a lock, where no operation in the critical section can begin till the lock is acquired. Similarly, we complete a release operation, only when all the operations before it have completed. This can be used to release a lock.

The last two models – ARM v7 and PowerPC – relax all orders including write atomicity. They thus allow for the maximum number of optimizations at the level of the compiler and architecture. They do have synchronization instructions though that enforce strict orders between synch instructions and the rest of the instructions.

Note that relaxing orders beyond a certain point is not necessarily a good thing. It can make the design of software more complicated. We might have to insert a lot of synchronization instructions and fences to make the code behave in a certain way. This has its performance implications. These issues will be dealt with in Section 9.6.

9.5.8 Summary

Let us now summarize our discussion. A memory model, MM, is characterized by the orderings that it respects. It needs to respect ws and fr because of PLSC. Then it needs to respect a subset of po and rf. Let it respect $gpo \subseteq po$ (program orders) and $grf \subseteq rf$ ($write \rightarrow read$ orders).

We thus can write,

$$MM = (gpo \cup ws \cup fr \cup grf) \tag{9.10}$$

9.6 Data Races

We discussed lock and unlock functions in Section 9.4.7. These are required to ensure that parallel code executes correctly. Otherwise, we will not be able to implement critical sections, which are required to correctly execute parallel code. Furthermore, in Section 9.4.7, we had considered an example where parallel threads try to update an account's balance. We concluded that two concurrent updates to the *balance* variable of the same account might lead to an incorrect state. To fix this situation, we had decided to enclose the update in a critical section.

9.6.1 Critical Sections, Concurrency Bugs, and Data Races

Now, if we forget to use critical sections, will we always have an error? The answer is NO. Let us look at an example.

```
counter++;
```

In this case, we are just incrementing a global counter. The correctness of this piece of code depends on how it is implemented in assembly (see the code snippets below). If we implement it as three instructions, where we first read the value of the *counter* from memory, increment the value that has been read, and then write the value to the memory location that holds the variable, *counter*, then there is a possibility of an error, a concurrency bug. This is because another concurrent update operation can also read the same initial value of the *counter* variable. This will lead to one update getting lost. However, if this statement is mapped to a fetch-and-increment atomic operation, then there is no possibility of an error because it is an atomic operation – not a *regular* read or write. Both the updates to the counter will get reflected in the final state of the program.

Multiple instructions

```
t1 = counter;
t2 = t1 + 1;
counter = t2;
```

Single instruction

```
fetch_and_increment(counter);
```

Note that in these code snippets $t1$ and $t2$ represent temporary variables that are mapped to registers. The first example with multiple instructions might clearly lead to incorrect execution, whereas the second example will not. Before proceeding further, we need a far more precise definition of what is an error in a parallel program, and how do we deal with it.

In general, in a parallel program, if we run it multiple times, the order of operations will be different because of the complex interplay of messages in the NoC and the memory system. However, we want the parallel program to be correct in all cases. For example, if it is multiplying two matrices, then the result should always be correct irrespective of the order in which the instructions are executed. To ensure that this genuinely does happen we need to regulate the behavior of concurrent accesses to the same variable. If both are reads, then there is no problem. However, if at least one of them is a write, then there is a problem; the order of accesses to the variable become important. Different orders might lead to different outcomes. A pair of accesses to the same address where at least one of them is a write are said to be *conflicting accesses*.

Definition 83
A pair of accesses to the same address, where at least one of them is a write, are said to be conflicting accesses.

Let us reconsider our example with the *counter* variable. If it is implemented with regular load/store assembly instructions, then there is a possibility that the execution might be incorrect. This is because it uses regular reads and writes, and this is where there is a possibility of an error because of concurrent and conflicting accesses by the two threads. Let us characterize this scenario by defining the term *data race*. A *data race* is informally defined as a situation where we have regular, concurrent, and conflicting accesses to a variable by different threads, where at least one of them is a write access. If we can eliminate data races in our program, then we can at least claim that between any two conflicting accesses to the

same variable, there is some kind of an order between them. Such an *order* must have been enforced by the programmer using program logic and synch instructions. This order ensures that the accesses are ordered sequentially and such kind of errors do not happen. This means that one thread will finish its updates, and then somehow signal another thread to begin. If we were to enclose the counter update function within lock and unlock functions, then such an order will automatically be imposed. It will not be possible to incorrectly update the *counter* variable. However, we can always make concurrent and conflicting accesses using synchronization operations such as *fetch_and_increment*. They update the *counter* variable atomically and correctly.

We have deliberately not defined the term *concurrent accesses* precisely. In computer architecture parlance, it does not mean "at the same time". It has a deeper meaning, which we shall explore in the subsequent sections.

From our informal discussion, we have learned several things. Two conflicting accesses need to be somehow *ordered* if we are using regular loads and stores. Otherwise, the output of the program may be wrong as we saw with the example to update the *counter*. This order can be enforced by wrapping the code in a critical section (demarcated by a lock and unlock function), otherwise we need to use atomic operations such as *fetch_and_increment*. Let us formalize this.

9.6.2 Data Races in the Context of Memory Models

A Formal Model of Data Races

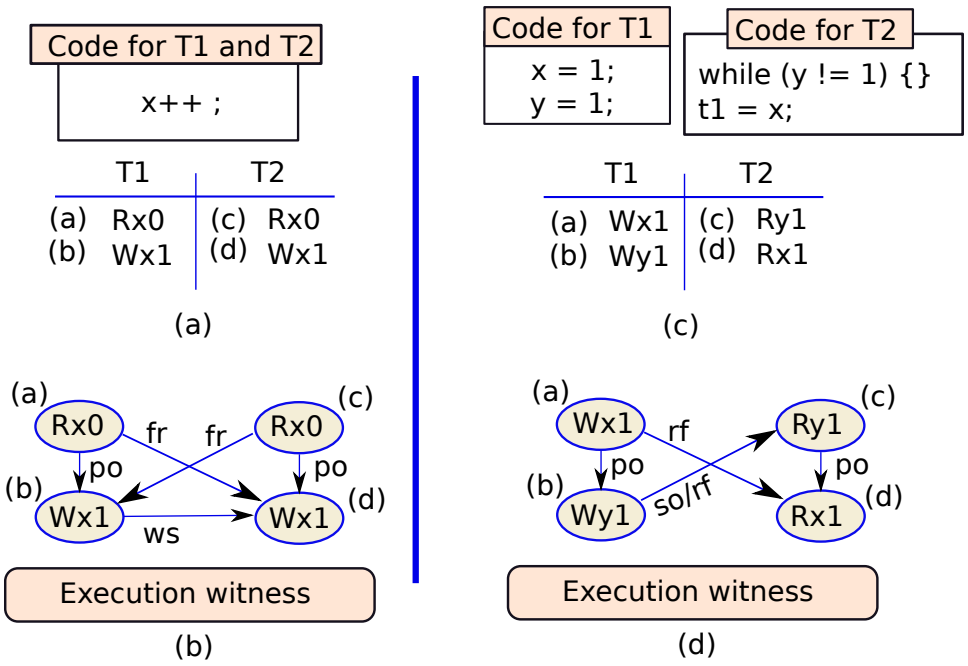

Figure 9.57: (a) and (b): Code and execution witness of a program that updates a counter. (c) and (d): Code and execution witness of a program that transfers the value of variable x across threads. Assume that y is a synch variable.

Assume an SC machine. Consider the code and execution witness in Figure 9.57 (a) and (b). It shows the code of a regular counter update where there is a data race (as we have defined, albeit, informally). The value of x (counter in this case) is finally set to 1, which is wrong. We need to disallow such executions. Now consider the code in Figure 9.57(c) and (d). Further, assume that x is a regular

variable and y is a synch variable, and the *while* loop exits in the first iteration. This piece of code basically transfers the value of x from thread T1 to thread T2. This execution seems to be correct. Until the value of y has not been read to be 1 by T2, it will keep looping. Once it reads $y = 1$, it has to read $t1 = x = 1$. This is correct execution on an SC machine. In fact all executions will yield the same output, which is correct.

So what is wrong in the execution witness shown in Figure 9.57(b) and what is correct in the execution witness shown in Figure 9.57(d)? Look closely. Consider only regular variables: x in both cases. The answer is that between two conflicting accesses in Figure 9.57(b), there is a path that has no synchronization order (*so*) edges. For example, between the instructions (b) and (d) (both $Wx1$), we only have a path with a *ws* edge. Now, focus on the execution witness shown in Figure 9.57(d). The path from $Wx1$ to $Rx1$ has an *so* edge. This is the crux of the definition of *concurrent accesses*. Consider two accesses to the same regular variable in the execution witness. If there is no path between them with an *so* edge, they are said to be *concurrent*. Let us now define a data race with the concepts we have just learned.

Definition 84
Consider two accesses to the same regular variable across threads. If there is no path between them in the execution witness with an so edge, they are said to be concurrent. *Whenever we have a pair of such conflicting and concurrent accesses, we refer to this situation as a* data race.

Let us appreciate the definition. We want that at least one path should exist with an *so* edge between conflicting accesses to the same variable, and this edge should be across threads. When we had such an edge, we saw that the execution was correct, and the lack of such an edge led to an incorrect execution. Can we generalize this?

When there is a path with an *so* edge, it means that synchronization instructions of the program are involved in enforcing a dependence between two conflicting accesses to the same variable. It will allow us to regulate conflicting accesses. Of course, here we need to differentiate between regular and synchronization variables. We do allow concurrent and conflicting accesses to synchronization variables: we assume that they are always updated in a sequentially consistent fashion. However, when it comes to regular variables, if we want the program to be free of data races, then if there is an rf, fr, or ws edge between any two operations on regular variables in the execution witness across threads, there has to be another alternative path between them that has *so* edges. This would mean that the operations are ordered by other instructions; they are not *concurrent*.

Does SC Imply Data-Race-Freedom?

Let us look at the examples shown in Figure 9.58.

We observe in Figure 9.58 that for the same code, we can have many sequential executions on an SC machine. The first execution (Figure 9.58(a)) is free of data races because there is a happens-before relationship between $Rx0$ and $Wx1$ with an *so* edge; however, the second execution has a data race because there is no happens-before ordering between the accesses $Wx1$ and $Rx1$ with an *so* edge. Now from our point of view, this *code* has a concurrency bug because it is possible to have an execution that has a data race. Hence, SC does not guarantee data-race-freedom.

Does Data-Race-Freedom Imply SC?

Let us ask the reverse question now. Assume an execution does not have any data races. Is it in SC? Let us look at a few theorems.

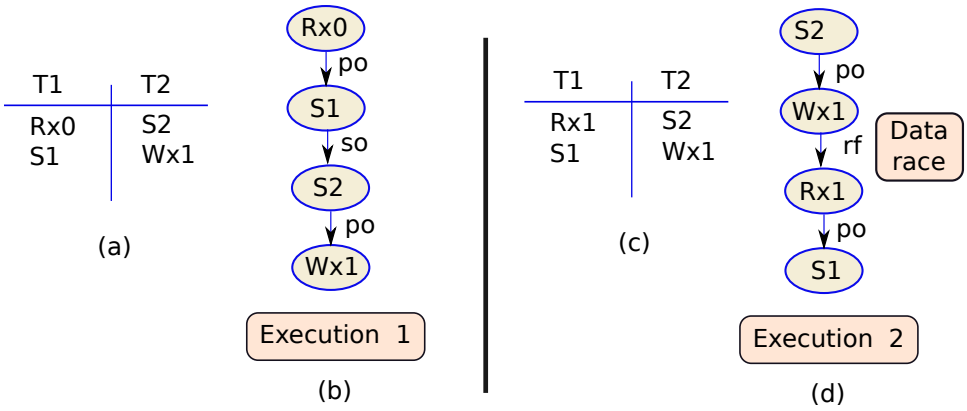

Figure 9.58: Different SC executions of the same program. $S1$ and $S2$ are synchronization operations.

Theorem 9.6.2.1 *Consider two conflicting accesses e_1 and e_2 in two different threads T_1 and T_2, where $e_1 \overset{hb}{\rightarrow} e_2$. If the execution is data-race-free, then there have to be two synchronization operations s_1 and s_2 with the following properties.*

- $s_1 \in T_1$ *and* $s_2 \in T_2$
- $e_1 \overset{hb}{\rightarrow} s_1$, $s_2 \overset{hb}{\rightarrow} e_2$, *and* $s_1 \overset{hb}{\rightarrow} s_2$
- *The paths from $e_1 \rightarrow s_1$ and $s_2 \rightarrow e_2$ will each have a po edge.*

Proof: Given that the execution is data-race-free, there will always be a path from e_1 to e_2 that has at least one *so* edge in the execution witness. Let us name this path $\mathcal{P}$. Assume that there are two threads in the system: T_1 and T_2.

By the definition of data-race-freedom, in the path $\mathcal{P}$, we will find two accesses s_1 and s_2 to a synch variable. This will be an *so* edge.

Given that we will find such an edge from $s_1 \in T_1$ to $s_2 \in T_2$, we see that we have satisfied all the conditions. We have $e_1, e_2, s_1, s_2 \in P$. From the definition of a synch operation, it follows that $e_1 \overset{hb}{\rightarrow} s_1$, $s_1 \overset{hb}{\rightarrow} s_2$, and $s_2 \overset{hb}{\rightarrow} e_2$. Note that since e_1 and s_1 access different addresses, there has to be a *po* edge between them. The same holds for s_2 and e_2.

This argument can easily be extended to the case of multiple threads. ∎

Now, the time has come to expose the magic of data-race-freedom. We shall prove that it implies SC execution regardless of the memory model.

Theorem 9.6.2.2 *A data-race-free execution is in SC regardless of the memory model.*

Proof: Consider a data-race-free execution E. Let us add all the edges to the execution witness that an SC execution needs to have namely *po*, rf, fr, and ws edges. If there is no cycle, the execution is in SC. Assume there is a cycle.

First consider the case of two threads. There have to be at least two edges in E with the following properties. The first edge has to be from $e_1 \in T_1$ to $e_2 \in T_2$. The second edge has to be from $e_3 \in T_2$ to $e_4 \in T_1$. Without loss of generality, assume that the cycle is of the form: $e_1 \overset{hb}{\rightarrow} e_2 \overset{hb}{\rightarrow} e_3 \overset{hb}{\rightarrow} e_4 \overset{hb}{\rightarrow} e_1$. As proven in Theorem 9.6.2.1, for any edge of the form $e_1 \overset{hb}{\rightarrow} e_2$, we need to have an edge of the form $s_1 \overset{hb}{\rightarrow} s_2$, where $e_1 \overset{hb}{\rightarrow} s_1$, and $s_2 \overset{hb}{\rightarrow} e_2$. $s_1 \in T_1$ and $s_2 \in T_2$. s_1 and s_2 are synch instructions. Similarly, we will have $s_3 \overset{hb}{\rightarrow} s_4$, where $e_3 \overset{hb}{\rightarrow} s_3$ and $s_4 \overset{hb}{\rightarrow} e_4$. Here, $s_3 \in T_2$ and $s_4 \in T_1$.

Since we have a cycle comprising $\langle e_1, e_2, e_3, e_4 \rangle$, and program orders hold between synch operations issued by the same thread, we shall also have a cycle comprising the accesses $\langle s_1, s_2, s_3, s_4 \rangle$. However, we have assumed that synch instructions' execution is sequentially consistent. Hence, they cannot form a cycle. This proves by contradiction that we cannot have a cycle in the execution witness between e_1, e_2, e_3, and e_4. This result can be extended to consider multiple threads. Hence, the execution is in SC. ■

Herein lies the greatness of data-race-freedom – it implies SC. Let us quickly recapitulate what we have proven.

Property	Reference
SC does not imply data-race-freedom	Figure 9.58
Data-race-freedom implies SC	Theorem 9.6.2.2
Non-SC execution implies data races	Contrapositive of Theorem 9.6.2.2
What do data races imply?	–

Let us now see what does having data races imply? Let us say that we have data races in a given execution with a certain memory model. Can we say something more? It turns out that we can. See the following theorem.

Theorem 9.6.2.3 *If we have a data race in a program, then it is possible to construct a sequentially consistent execution that also has a data race.*

Proof: Assume a multithreaded program has an execution, E, that exhibits a data race. This execution is as per the memory model of the machine. Let us construct an SC execution from it that also has a data race.

Let us keep running the program until we detect the first data race. Assume that just after executing the memory operation e_j, we observe the first data race. We stop there. Let us refer to this partial execution as $\hat{E}$. Till this point $(\hat{E} - e_j)$, the execution has been data-race-free. By Theorem 9.6.2.2, the execution $\hat{E} - e_j$ is sequentially consistent because it is free of data races. Let us now add e_j to the execution.

Is execution $\hat{E}$ still sequentially consistent? Assume it is not. Then there will be a cycle involving e_j. Let $e_j \in T_j$ and let the cycle be of the form $e_{j-1} \overset{hb}{\rightarrow} e_j \overset{hb}{\rightarrow} e_1 \ldots \overset{hb}{\rightarrow} e_{j-1}$. If the cycle has any other node that is in thread T_j, then we need to have a synchronization edge, because before adding e_j, no data races were detected. If the synch operation in thread T_j is after e_j in program order, then it should not have executed in the first place because e_j had not completed. This is a contradiction. If it is before e_j in program order, then also we cannot have a cycle involving e_j because there will be a path from e_1 to e_j containing *so* edges. This means that e_j is globally ordered after e_1 and there can be no path from e_j to e_1. Hence, the only option is that there are no other nodes of the cycle in T_j.

Assume e_j is a write. Note that because SC has held up till now, no node in $\hat{E} - e_j$ has read the value written by e_j. Hence, we can treat e_j as the latest write to its location. Thus, no rf or ws edges will emanate from it, and it cannot complete a cycle.

Now, assume that e_j is a read. This means that an rf edge will enter it, and an fr edge will exit it. Let the rf edge be from e_i to e_j, and let the fr edge be from e_j to e_k. By definition, we will also have a ws edge from e_i to e_k. For the cycle to complete, there needs to be a path from e_k to e_i that does not have e_j. This means that there will be a cycle that does not involve e_j. This would have existed even before e_j was considered. Given that SC held up till now, this is not possible. Hence, there is a contradiction in this case as well.

Hence, there are no cycles and the execution $\hat{E}$ is in SC. Furthermore, this execution has a data race. Let the equivalent sequential order be $\mathcal{S}$.

Is the execution $\hat{E}$ complete? This means that if a given operation of a thread is present, are all of its previous operations (in program order) there? If they are not there, let us add them. Let us refer to all the operations that are missing as the set of *skipped operations*. For every such *skipped operation*, there is some memory operation $e \in \hat{E}$ that succeeds it in program order. Because $\hat{E}$ is in SC, the backward slice (all the operations that determine the values of the operands) of every operation in $\hat{E}$ is present in it. Furthermore, because of PLSC all the preceding instructions of $e \in \hat{E}$ in the same thread that access the same address are also present in $\hat{E}$. This means that adding the skipped instructions is not going to change the outcome of memory operations in $\hat{E}$.

Now, let us add the skipped instructions to the equivalent sequential order, $\mathcal{S}$. Note that given that we are at liberty to set their outcome and moreover their outcome does not influence the values read or written in $\hat{E}$. For each thread we add its earliest skipped instruction at the appropriate point (as per program order) in $\mathcal{S}$. It reads the latest value in the sequence. Similarly, for a write, we also add it at an appropriate point in $\mathcal{S}$. We assume it is the latest write for the location. Given that we are ensuring that the resultant sequence of instructions is in SC after each step, we can prove by induction that after adding all the skipped instructions, the final sequence is still in SC, and still has data races.

We can then simulate the rest of the execution in a sequentially consistent manner.

This proves that it is possible to construct an SC execution that also contains a data race, if the original program has a data race with any memory model.

■

From the results of Theorem 9.6.2.3, we can say that **a program has a data race, irrespective of the memory model**. We can complete the table now.

Property	Reference
SC does not imply data-race-freedom	Figure 9.58
Data-race-freedom implies SC	Theorem 9.6.2.2
Non-SC execution implies data races	Contrapositive of Theorem 9.6.2.2
A program with a data race has an SC execution with a data race	Theorem 9.6.2.3

9.6.3 Properly Synchronized Programs

Can we guarantee that all executions of a program are data-race-free? Such a program is called a data-race-free or a DRF program.

Let us first answer this question. Consider a program where every shared variable is accessed within a critical section, and the same shared variable is always protected by the same lock. Recall that a critical section is demarcated by lock and unlock functions – these functions access synch variables. This automatically disallows concurrent accesses to the same shared variable because only one thread can

hold a lock at a time, and thus it is not possible for two threads to concurrently access the same shared variable – one of them will not be able to acquire the corresponding lock for the shared variable. This program is thus data-race-free. Let us refer to such programs as *properly synchronized programs* or PS programs.

Definition 85 *In a properly synchronized program (PS program), every shared variable is accessed within a critical section, and throughout the program, the same shared variable is protected by the same set of locks. This ensures that we cannot have concurrent accesses by two threads to the same shared variable. Such programs are free of data races.*

Discussion

We thus observe that any properly synchronized program is data-race-free and always produces sequentially consistent executions. It is thus a DRF program. In other words, properly synchronizing a program ensures that our executions are both data-race-free and in SC regardless of the underlying memory model! This is arguably one of the most impactful results in modern parallel computing and parallel architecture, and allows hardware designers to pursue all kinds of performance enhancing optimizations while maintaining the intuitiveness of the high level code.

Regardless of the memory model, all that programmers need to do is that they need to enclose all the accesses to shared variables in critical sections (individually or in groups), and always ensure that the same shared variable is protected by the same set of locks. Once this is done, the execution is in SC, and thus it is very easy to write parallel programs. Additionally, our executions do not exhibit data races, as a result we avoid many classes of concurrency bugs.

Regarding performance, this depends on the proportion of shared variables that are accessed. In most modern parallel programs, shared variable accesses are relatively infrequent. Most of the accesses are to private data (private to a thread), therefore there is no additional overhead in terms of synchronization instructions while accessing such data. Given this pattern, the overheads of properly synchronizing are considered to be rather modest, and it is by and large possible to reap the advantages of a relaxed memory model.

The main challenge now is to ensure that a given program is properly synchronized. This is unfortunately computationally undecidable, and thus it is not possible to write a tool to find this out. However, we can analyze programs and their executions for evidence of data races. If we find a data race, we can conclude that the program is not properly synchronized, and we can also pinpoint the regions of the code the programmer should look at based on the addresses involved in the data race. Note that the absence of data races in a few sample runs does not indicate that the program is properly synchronized, however, this approach has proven to be an extremely efficient and successful method for finding bugs in parallel programs.

9.6.4 DRF Memory Models

The main aim is to "properly synchronize" a program. This will give us the best of all worlds: relaxed memory models, SC execution, and data-race-freedom.

Now how do we do this? As discussed, one way is to enclose all accesses to shared variables within critical sections. This means that the high level language needs to give us the facility for creating critical sections. We can thus define a memory model at the level of a high level language such as C++ or Java that specifies a set of orderings similar to memory models in hardware. Additionally, this model needs to specify what a programmer needs to do to produce data-race-free executions. Such memory models are known as DRF memory models. They are defined for programming languages.

They can be of different kinds because we can have many kinds of synch operations. For example, a synch operation need not be a regular fence operation. We can instead use the acquire and release operations defined in release consistency (see Section 9.5.7). Theorem 9.6.2.1 only says that the following relations need to hold: $e_1 \overset{hb}{\rightarrow} s_1$, $s_2 \overset{hb}{\rightarrow} e_2$, and $s_1 \overset{hb}{\rightarrow} s_2$. s_1 and s_2 need not be regular fence operations; s_1 can be a release and s_2 can be an acquire. The theorem will still hold. We can then prove that with such acquire and release operations, data-race-freedom implies SC. A DRF model that provides such acquire and release operations will be different from a DRF model that just provides regular fences. Modern languages such as C++and Java have many such synchronization constructs and thus provide a complex DRF model.

9.6.5 Lock Set Algorithm

We have concluded that by ensuring that there are no data races, we will not face issues with different memory models, particularly models that have non-atomic writes. All that we need to ensure is that the code is properly synchronized and consequently data-race-free. Even though such theoretical models have taken us very far, still in practice, it is possible that programmers might forget to "properly synchronize" their programs. In this case, we might encounter data races, and this will give rise to a new class of bugs called *concurrency bugs*.

Hence, there are several algorithms in both software and hardware to ensure that the code is properly synchronized. Let us discuss one of the simplest algorithms in this class known as the *Lock Set Algorithm* [Savage et al., 1997]. The basic idea is as follows. We associate a set of locks with each memory location v (the lock set). When we initialize the program, we assume that the lock set for each location contains all the locks that the program uses.

Additionally, each access (read or write) is also associated with a lock set. It is defined as a set of locks held by the thread that is issuing the memory access.

To summarize, there are two lock sets that we are considering: one held by the memory location ($L(v)$), and the other held by the thread, T, while accessing the memory location (represented as $L(T)$). Now it is possible that several locks protect a given memory location, however we do not know them. Hence, we need to instrument reads and writes to the shared variables, and find the set of locks that protect each shared variable (its lock set). Hence, after each access we compute

$$L(v) = L(v) \cap L(T) \tag{9.11}$$

This further narrows down the set of locks that protect each variable. In every step, we keep iterating and refining the lock set till we reach the end of the program. At this point of time, there are two possible scenarios. The first scenario is that the lock set is non-empty. This means that for the variable (and its associated memory address), we have found a set of locks that protect it. The other scenario is when the lock set is empty. In this case, it means that we were not able to find a set of locks that protect a given location. This means that most likely there is a data race associated with this location.

The standard approach for implementing this technique is in software. In this case, we augment each access to a shared variable with additional code that refines the lock set for the variable. At the end, we analyze the lock sets for all the shared variables, and find those variables with empty lock sets. We can then report these variables to the programmer, and then she can check if the code that accesses these variables is properly synchronized or not. This is thus a method for *data race detection*.

This basic mechanism is however suboptimal as pointed out by Savage et al. [Savage et al., 1997]. There are three important patterns in modern programming languages that are falsely reported to be data races, whereas these patterns are perfectly safe.

Initialization We often initialize shared variables without using any locks.

Read-only variables A program might use many read-only variables that are written once during initialization, and later they are read many times. This algorithm will report such accesses to be data races. This is however not the case.

Reader-writer pattern Such access patterns allow multiple readers to read the same variable concurrently. Multiple concurrent read accesses do not lead to data races. However, this algorithm will find the different read accesses to contain different lock sets, and thus might report a few of them to be data races.

It is thus necessary to refine the basic algorithm.

Improved Version of the Basic Lock Set Algorithm

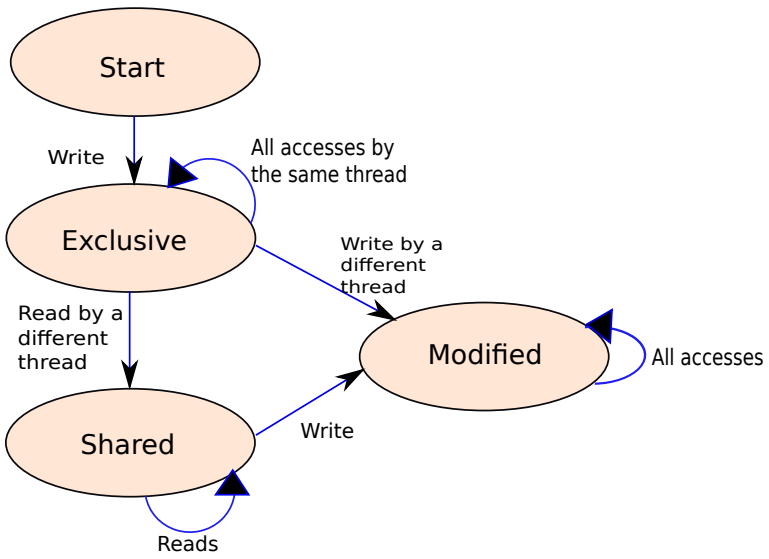

Figure 9.59: States in the advanced version of the lock set algorithm

Figure 9.59 shows the state diagram for the advanced version of the algorithm. We start in the *Start* state. The first access to a memory location has to be a write because we need to set its initial value. After it is initialized with the write, the initializing thread will continue to make accesses (reads or writes). This should be kept out of the purview of the data race detection algorithm. We refer to this state as the *Exclusive* state. Subsequently, we need to track the accesses made by other threads.

If there is a read by another thread, we transition to the *Shared* state. This captures the multiple-readers scenario. At this point, even if multiple threads are reading the variable, we do not report a data race. However, if there is a write, then we need to start using the regular lock set-based data race reporting algorithm. We transition to the *Modified* state. On similar lines, if we have a write access by another thread in the *Exclusive* state, we transition to the *Modified* state. Subsequently, we remain in this state and keep using the regular lock set algorithm, where we continue refining the lock set. Only in this state, we detect and report data races.

9.6.6 Data Race Detection with Vector Clocks

Let us now discuss one more approach that uses a mechanism called vector clocks. Most approaches in this space are in software, where a compiler or a module at runtime, instruments the reads and writes to record their dependences. If we find some unusual patterns symptomatic of data races, then an error is flagged. There are approaches in hardware to detect dynamic data races; however, they are expensive and at best only make probabilistic guarantees.

Theoretical Preliminaries

Let us now look at a more general mechanism for detecting data races.

The key question that we need to answer is how do we find if two events are concurrent? Recall that in the world of concurrent systems, two concurrent events need not take place at the same time. In fact, when we are considering multiple cores, with their own clocks, the definition of time itself is fuzzy. We need to come up with an alternative definition.

Till now, we have been saying that two events e_1 and e_2 are concurrent if there is no happens-before ordering between them. We have however not dwelt on how to find if there is a happens-before ordering between two events. To do so, we will use results from classic distributed systems literature – vector clocks.

Consider n processes. There is no global clock. All the processes have their separate clocks, and the relationship between these clocks is not known. There are two kinds of events: internal and external. Internal events are local to a process. They are not visible to other processes. However, external events are visible to other processes. They are modeled as send-receive messages, where one process sends a message to another process.

Let every process contain an n-element vector, which is a *vector clock*. The i^{th} process's vector is denoted as V_i. The i^{th} element of V_i represents the local time of process i. We increment the local clock, $V_i[i]$, before sending a message, and after receiving a message. $V_i[j]$ $(i \neq j)$ represents i's best estimate of j's local time.

Let us see what happens when processes send and receive messages. Before sending a message, process i increments $V_i[i]$, and sends the message along with its vector clock, V_i. When j receives the message, it first increments its own time, $V_j[j]$, and then sets V_j as follows.

$$\forall k, V_j[k] = max(V_i[k], V_j[k]) \tag{9.12}$$

We refer to this as the *union* operation of V_i and V_j: $V_i \cup V_j$.

Here, the symbol $\forall$ stands for *"for all"*. This operation gives j the most up-to-date estimate of the local times of the rest of the processes. Given two vector clocks, we can define a few relationships between them. Two vector clocks, V_i and V_j, are equal if and only if all their elements are pairwise equal.

$$V_i = V_j \Leftrightarrow \forall k, V_i[k] = V_j[k] \tag{9.13}$$

Let us now look at the conditions where we can say that event e_i (by process i) happened before event e_j (by process j). Let us define a precedence relationship, $\prec$, between vector clocks. Let us say that $V_i \prec V_j$ if the following relationship holds (note that $\wedge$ stands for logical AND).

$$V_i \prec V_j \Leftrightarrow (V_i \neq V_j) \wedge (\forall k, V_i[k] \leq V_j[k]) \tag{9.14}$$

The first term in Equation 9.14 means that the two vector clocks are not equal (both the arrays are not exactly equal). The second term means that each entry in the first vector clock V_i is less than or equal to the corresponding entry in V_j. This alternatively means that there is some k' for which $V_i[k'] < V_j[k']$. Let us understand the logic behind comparing vector clocks in this manner.

Causal Ordering

Let us list some classic results in distributed systems. Let event e_i happen at time V_i and event e_j happen at time V_j.

Theorem 9.6.6.1 $V_i \prec V_j \Rightarrow e_i \overset{hb}{\rightarrow} e_j$

Theorem 9.6.6.2 $e_i \overset{hb}{\rightarrow} e_j \Rightarrow V_i \prec V_j$

By considering both the theorems, we can say that the following relationship holds.

$$e_i \overset{hb}{\rightarrow} e_j \Leftrightarrow V_i \prec V_j \tag{9.15}$$

Two events, e_i and e_j, are said to be concurrent if none of the following relationships hold: $V_i \prec V_j$ or $V_j \prec V_i$. We write $V_i || V_j$ or $e_i || e_j$ to indicate concurrency.

Definition 86 *A vector clock is defined as an n-element vector, where there are n processes in the system. Whenever, process i sends a message to process j, it also attaches its vector clock along with the message. The vector clock of each process is initialized to all zeros.*

Before process i sends a message, it increments, $V_i[i]$. When process j receives the message, it first increments its own time, $V_j[j]$, and then sets V_j as follows.

$$\forall k, V_j[k] = max(V_i[k], V_j[k])$$

Two vector clocks can be compared with a precedence relationship.

$$V_i \prec V_j \Leftrightarrow (V_i \neq V_j) \wedge (\forall k, V_i[k] \leq V_j[k])$$

For vector clocks, we have the following relationship.

$$V_i \prec V_j \Leftrightarrow e_i \overset{hb}{\rightarrow} e_j$$

Two events, e_i and e_j, are said to be concurrent if none of the following relationships hold: $V_i \prec V_j$ or $V_j \prec V_i$.

Data Race Detection using Vector Clocks

If we have n threads, we assign an n-element vector clock to each thread (*process* in theoretical parlance). Additionally, each memory location, v, is assigned two vector clocks: a read clock R_v and a write clock W_v. Let C_T be the vector clock of the current thread, *tid* be its thread id, and let C_L be the vector clock associated with the acquired lock.

Algorithm 1: Lock acquire

```
1 C_T[tid] ← C_T[tid] + 1
2 C_T ← C_T ∪ C_L
3 C_L ← C_T
4 C_T.inLock ← True
```

Let us first consider the lock acquire function. In this case, we are using synch variables, and since the system ensures an SC execution for such variables, we have allowed data races between their accesses. Whenever, a given thread acquires a lock, it is necessary to set both the vector clocks to the same time because this point is a rendezvous point for the thread and the lock. Hence, we first increment the local clock $C_T[tid]$ of the current thread, compute the union of both the vector clocks (C_T and C_L), and set

both of them to the computed union. Finally, we set the *inLock* bit of the current thread to 1, which indicates the fact that the current thread is inside a critical section.

Algorithm 2: Lock release

```
1 C_T.inLock ← False
```

On similar lines, when we release the lock we set the *inLock* bit to 0. A thread may make accesses to shared variables without a lock, some of these will be data races.

Algorithm 3: Read operation

```
1 if ¬C_T.inLock then
2 |   C_T[tid] ← C_T[tid] + 1
3 end
4 if W_v ⪯ C_T then
5 |   R_v ← R_v ∪ C_T
6 end
7 else
8 |   Declare Data Race
9 end
```

Let us now discuss the read operation. If the current thread does not hold a lock, then an access is being made outside a critical section. We are not in a position to detect if this variable is shared or not. However, to indicate that this is a separate event, we increment $C_T[tid]$ (local clock of the current thread).

For any read operation, all the writes to location v should precede it or be equal (denoted by the $\preceq$ symbol) in terms of logical time. We explicitly verify this by comparing the write clock W_v with the current time. If W_v precedes the current time or is equal to it, then we replace the read clock R_v with $R_v \cup C_T$. This ensures that the read clock is up-to-date as per the semantics of standard vector clocks. Note that we do not require the read clock to precede or be equal to the current time because we allow concurrent reads in our system – they are not classified as data races.

However, if we find that $W_v || C_T$ or $C_T \prec W_v$, then there is a data race, and it is immediately flagged.

Algorithm 4: Write Operation

```
1  if ¬C_T.inLock then
2  |   C_T[tid] ← C_T[tid] + 1
3  end
4  if (W_v ⪯ C_T) ∧ (R_v ⪯ C_T) then
5  |   R_v ← R_v ∪ C_T
6  |   W_v ← W_v ∪ C_T
7  end
8  else
9  |   Declare Data Race
10 end
```

Finally, let us consider the write operation. Here also, we first check if the access is made within a critical section or outside it. This is handled on the same lines as the read operation.

For a write, we need to ensure that both the read clock and the write clock either precede or are equal to the current time. This follows from the way we have defined data races. Writes need to be totally ordered with respect to prior reads and writes. If this is not the case, then we can immediately flag a data race. Otherwise, we proceed to update the values of R_v and W_v with information contained in the current time using the union operation.

This notion of vector clocks can thus be very easily used to create data race detectors in software.

9.7 Transactional Memory

We had discussed critical sections in Section 9.4.7, and then connected them to data races in the previous section. It should not be possible for other threads to see the values written by instructions in the critical section before it has ended, and the lock has been released. Also, instructions within the critical section should not be able to see values written by concurrent writes. This means that the entire block of code within the critical section needs to appear to execute as a single statement. Either it appears to other instructions that the entire block of code has executed, or it appears that the execution of the block has not begun. This property as we have seen in the case of write operations is known as *atomicity*. It basically signifies an all-or-none behavior. We need atomicity at the level of critical sections as well.

Let us now also consider how we implemented critical sections. We created a special pair of *lock* and *unlock* functions, where the lock function uses an atomic exchange instruction to atomically exchange the contents of a register and a memory location. This instruction does the atomic exchange in such a manner that no other instruction can interrupt it in the middle or can view any intermediate state. The atomic exchange instruction is supported by almost all instruction sets as of 2020, and more sophisticated locking algorithms use other primitives such as atomic fetch-and-increment, and atomic compare-and-set. All of these atomic primitives are implemented by modifying the coherence protocol. We delay sending acknowledgements to the directory till the atomic operations complete.

Such simple modifications to the coherence protocol allow us to implement atomic instructions, which in turn allow us to implement critical sections in parallel programs. All parallel programming libraries including *pthreads* and *OpenMP* implement a variety of locking mechanisms that allow us to create many kinds of critical sections. Additionally, they ensure fairness by providing a guarantee on how long it will take a thread to acquire a lock.

However, there are problems with such conventional mechanisms. Consider the following C code snippet to credit or debit money from a bank account referred to as *account*. In this case, the bank account is an instance of a class of type *Account*. Let the *Account* class have a single field called *balance*. The way we present the following code snippet is similar to the way we presented the notion of critical sections in Section 9.4.7, where each line corresponds to a line of assembly code.

Listing 9.2: Java code to update the balance in a bank account. Assume that *account* is passed by reference (a pointer to it is passed).

```
void updateBalance(int amount, Account account){
    lock();

    int temp = account.balance;
    temp = temp + amount;
    account.balance = temp;

    unlock();
}
```

In this case, we lock all three lines. However, if a bank has a lot of accounts, then it is not necessary that two accesses to the *updateBalance* function access the same data. In fact, they might be accessing different sets of data (different accounts). There will be no overlap in terms of memory addresses between the two sets of accesses; however, given the nature of our critical section, we will only allow one of the threads to proceed. In other words, with conventional locks, we do not allow *disjoint access parallelism*. This means that if different threads access different accounts, we do not execute them in parallel. In the conventional code that we show, irrespective of the data being accessed, we encapsulate the statements accessing shared data in a critical section, and force threads to execute the critical section in sequence. This is good because it ensures correctness; however, it is bad because it limits opportunities for parallel execution.

Let us look at the term disjoint access parallelism in some more detail. It is defined as a property of a parallel program, where two threads can execute the same set of statements concurrently if they access different sets of data. A critical section as shown in Listing 9.2 that is enclosed between a lock and an unlock statement, does not allow disjoint access parallelism.

Definition 87
Disjoint access parallelism is defined as a property of a parallel program, where two threads can execute the same set of statements concurrently if they access different sets of data.

Now, to enable disjoint access parallelism, we can change the locking logic. Instead of having one single lock for the entire function, we can associate a different lock with each account. Before accessing an account, we need to acquire the lock associated with the account, and then once we are done with the processing, we can release the lock. The modified code is shown below.

```
void updateBalance(int amount, Account account){
    account.lock(); /* account specific synchronization */

    int temp = account.balance;
    temp = temp + amount;
    account.balance = temp;

    account.unlock();
}
```

Having a separate lock for each account takes care of the problem of an absence of disjoint access parallelism. However, it introduces other problems. Let us consider realistic code where we might be accessing different accounts. In this case, before executing the code of the critical section, we need to lock all the accounts that might be accessed beforehand. At the end we release all the locks. For example, if we want to write a function to transfer money from one bank account to the other, then we need to lock both the accounts.

Sadly, we may have a deadlock situation. Assume that there are two accounts A and B. It can so happen that similar to deadlocks in an NoC, we have a situation where thread 1 holds the lock for A and waits for the lock for B, and there is a reverse situation with thread 2. Then no thread will be able to progress, because there is a circular wait between threads A and B. A wants a resource that B has and at the same time B wants a resource that A has. Similar to ordering virtual channels, we can use the same algorithm here. If we acquire the locks in order, for example, if we acquire the lock for A before we acquire the lock for B, it is not possible to have a deadlock.

Again this approach creates a few more problems. This means that we need to be aware of all the locks that a given critical section is going to acquire at the beginning of the critical section. This might not always be possible in critical sections with a complex control flow. In fact, when the address of the account is computed dynamically, we might not be aware of the lock variable's address till we execute the relevant statements at runtime. We can always be conservative by prohibiting certain kinds of code within the critical section, particularly, code that dynamically computes the addresses of locks, and also ensure that we acquire a superset of locks at the outset – more than what we actually require.

Many such techniques unnecessarily restrict our freedom in writing parallel code and also reduce performance. It is thus essential to look at a solution beyond locks such that we can write critical sections with ease, and without bothering about how we acquire locks and avoid deadlocks. We can borrow inspiration from the world of database design and introduce the notion of *transactions*. A

transaction is defined as a block of code that executes atomically and allows disjoint access parallelism. This is exactly the property that we want, where in a certain sense the entire block of code executes as if it is a single instruction. Let us motivate our discussion by looking at how our running example will look like with support for transactions.

```
void updateBalance(int amount, Account account){

    atomic {  /* an atomic transaction */
        int temp = account.balance;
        temp = temp + amount;
        account.balance = temp;
    }
}
```

We create an *atomic* block where we assume that the code encapsulated within it executes atomically as a transaction – all or nothing. Moreover, it executes like a critical section and also allows disjoint access parallelism. This means that two instances of this code that access different variables can execute in parallel. The benefits of such an approach are obvious: ease of programmability and support for disjoint access parallelism. We need not bother about low level issues such as how locks are implemented.

Transactions have a notion of succeeding or failing. If a transaction succeeds, then it means that it was able to complete the execution of all the statements encapsulated within it. On the other hand, if there was interference from other threads meaning that different transactions clashed with each other by accessing the same set of addresses in a conflicting manner, the transaction might need to fail or *abort*. In this case, the transaction is said to have *failed* or *aborted*. In either case – success or failure – the transaction should appear to execute instantaneously; moreover, its partial state (before completing) should not be visible to other threads. Formally, a transaction is expected to possess the four ACID properties.

Atomicity: Either the entire transaction completes or if there is a problem (discussed later) the entire transaction fails. If a transaction fails, no traces of its execution are visible to the same thread or other threads. This is also known as all-or-nothing semantics.

Consistency: Let us define a valid state of a system as a state that has been created by following all the rules of program execution, coherence, and consistency. If the state of the system is valid before a transaction starts its execution, then the state is valid after the transaction finishes its execution. The transaction might either succeed or fail; irrespective of the outcome, it should appear that after the transaction is over, the state of the system is valid. For example, if a failed transaction leaves back some of its updates in the system, the state would be invalid. We need to ensure that this does not happen.

Isolation: Transactions are executed concurrently with other transactions and regular read/write instructions. Particularly with respect to other transactions, we wish to have a property akin to sequential consistency. The property of isolation states that a parallel history of transactions is equivalent to some sequential history of transactions, where transactions initiated by different threads execute one after the other. This further means that it appears that each transaction has executed in isolation.

Durability: Once a transaction finishes or *commits,* it writes its memory updates to stable storage. This means that those updates will not get lost.

Most transactional memory systems as of 2020 follow these four ACID properties. This ensures that each transaction looks like a large single instruction that executes atomically.

Definition 88
A transaction is defined as a block of code that executes atomically. It acts like a critical section; however, it also allows disjoint access parallelism. The transaction appears to execute instantaneously to other transactions. A software or hardware mechanism that has support for transactions is a transactional memory *system.*

A software-only mechanism is known as software transactional memory *(abbreviated as* STM*). On similar lines, a hardware based mechanism is known as* hardware transactional memory *(abbreviated as* HTM*).*

If a transaction executes successfully, then it is said to finish normally, and the finish operation is called a commit*. However, if it fails for some reason, then it is said to have* aborted*.*

9.7.1 Fundamentals of Transactional Memory

Let us now define some basic terminology with respect to transactions. Consider an atomic block of code. If we need to run it as a transaction, then we need support for transactions either exclusively at the level of software, or at the level of both hardware and software. At runtime, software or hardware entities will create transactions that will execute the instructions within an atomic block.

Conflicts

Consider a scenario, where we have two threads executing an atomic block. We thus have two transactions: TS_A and TS_B. If they access disjoint sets of variables, then they cannot affect each other's execution, and both the transactions can proceed in parallel. However, if there is an overlap in the set of variables that they access, then they are not executing in isolation. We would ideally like sequential consistency to hold among transactions, which means that they should appear to execute serially. Either TS_A sees the state written by TS_B or vice versa. However, if they are executing concurrently on different cores, and modifying the same set of variables, this will not happen. One of the transactions needs to be either stalled or it needs to abort.

If such a scenario arises, we say that the transactions are conflicting, or they have a conflict. A conflict is defined as follows. Let the set of variables that a transaction reads be defined as its read set ($\mathcal{R}$), and let the set of variables that a transaction writes be defined as its write set ($\mathcal{W}$). Let the read and write sets of transaction TS_A be $\mathcal{R}_A$ and $\mathcal{W}_A$ respectively. Similarly, let the read and write sets of transaction TS_B be $\mathcal{R}_B$ and $\mathcal{W}_B$ respectively.

We say that TS_A and TS_B *conflict* if and only if any one of the relations (shown below) is true.

$$\mathcal{R}_A \cap \mathcal{W}_B \neq \phi \tag{9.16}$$
$$\mathcal{W}_A \cap \mathcal{R}_B \neq \phi \tag{9.17}$$
$$\mathcal{W}_A \cap \mathcal{W}_B \neq \phi \tag{9.18}$$

In simple terms, if one transaction writes something that another transaction reads, then they have a conflict. Or, if two transactions write to the same variable, then also they have a conflict. However, if their read sets overlap ($\mathcal{R}_A \cap \mathcal{R}_B$), then this is not a conflict because we can read the same data in any order – it does not matter. At this stage, please realize that the notion of conflicts among transactions is similar to dependences between instructions. If two instructions have a dependence, they cannot execute in parallel.

Concurrency Control

With each conflict, we define three events of interest: occurrence, detection, and resolution. A conflict is said to have *occurred*, when the conflicting memory accesses happen (read-write, write-read, or write-write). It need not be *detected* immediately, it can be detected later. However, we are not allowed to detect a conflict after the transaction finishes – it will be too late. After we detect a conflict, we must *resolve* it. We can either roll back one of the conflicting operations and stall the transaction that issued it, or we can kill one of the conflicting transactions. Killing a transaction is also known as *aborting* a transaction. Sometimes aborting a transaction is the best choice, otherwise if we stall transactions, we might create a deadlock.

The timing of these events varies with the TM (transactional memory) system. It depends on the type of concurrency control, which is defined as the way in which we deal with accesses to shared variables. There are two common kinds of concurrency control: pessimistic and optimistic.

Pessimistic Concurrency Control In this type of concurrency control, conflict occurrence, detection, and resolution happen at the same point of time. In other words, this means that we do not execute any instructions after we detect a conflict, and this saves wasted work. However, we need to work harder to detect conflicts every time there is a memory access. In software-only schemes, this is hard to do; however, in hardware based systems, this approach does not have significant overheads.

Optimistic Concurrency Control In this case, we allow transactions to execute without performing a lot of checks while they are accessing memory variables. When a transaction completes, we check if it has completed successfully, and if there are any conflicts. If there are no conflicts, then the transaction commits, otherwise it aborts. This kind of concurrency control is well suited for software transactional memory systems, because this minimizes the work that needs to be done for each memory access. We simply need to check for conflicts at the end, which involves fewer instructions.

Conflict Detection

With pessimistic concurrency control, we detect conflicts as soon as they occur. This is also known as eager conflict detection. There are many flavors of eager conflict detection. We can either detect the conflict when the transaction accesses a variable or a cache block for the first time or when its coherence state changes, or we can detect conflicts every time the variable is accessed. The latter is a very inefficient approach, and thus is typically not preferred.

The other paradigm of conflict detection is lazy conflict detection, where we detect a conflict after it has occurred. This happens in systems with optimistic concurrency control. Here also, we have numerous approaches. We either detect a conflict at the time of committing a transaction, or we can detect it at specific points in the transaction known as *validation* points. The validation points can be inserted by the compiler, or can be decided dynamically by the hardware. At these validation points a dedicated thread or hardware engine checks for conflicts.

Conflict detection has some subtle complications. Assume that transactions TS_A and TS_B conflict. However, later TS_B gets aborted. It would have been wrong to abort TS_A based on the conflict, because its conflicting transaction TS_B ultimately got aborted. Hence, many systems have dedicated optimizations to take care of such cases.

Version Management

To ensure that a transaction executes in isolation, we need to ensure that all of its updates are not visible to other threads. This means that it needs to create a new *version* of memory for itself. This version of memory contains the values of all the variables/memory locations before it started, and the changes it has made to the variables in its write set. The changes that a transaction makes to memory is known as

the transactional state, and this needs to be made visible only after the transaction commits. Managing the transactional state is also known as version management.

Eager Version Management

There are two kinds of version management: eager and lazy. In eager version management, we directly make changes to memory. The write set is not separately buffered in any software or hardware structure. A thread goes ahead and changes the values of variables in memory, this reduces the read or write time. To safely recover the state if the transaction is aborted, we need to maintain an undo log. The first time that the thread writes to a variable in a transaction, it saves its previous value in an undo log, which can be a structure in hardware or software. Subsequent changes need not be logged because if there is an abort, we only need the value that existed before the transaction began.

Eager version management is very efficient for large transactions that abort very infrequently. All the updates to variables are directly sent to memory. Commits are fast because the changes are already there in memory. In comparison, aborts are very slow. We need to read all the values from the undo log and restore the memory state. However, the most important problem is maintaining isolation. If the values are written directly to memory, then other threads can see data written by the transaction before it has committed. This violates the property of isolation.

There are two ways of dealing with this problem. In software based systems, we add a version number to each variable in the write set. At commit time, we increment the version of each variable in the write set. Additionally, we lock each variable before writing to it. This ensures that other transactions cannot write to the variable at the same time. For transactions that read the variable, they record the version number, and also check the version number when they commit. If the version numbers do not match, then it means that some other transaction has written to the variable in the meanwhile. We shall discuss such schemes later in Section 9.7.3.

For hardware based systems, we augment each cache line with additional bits that indicate whether a variable has been read or written by an active transaction. The system does not supply the values of such variables to memory accesses made by other threads. This ensures isolation. Once a transaction commits or aborts, it is necessary to clear all such bits. There is a fast method in hardware to clear such bits. It is known as *flash clearing*. Flash clearing can be used to quickly set or unset a given bit in all the lines of a cache. We can then discard the undo log.

If a transaction with eager version management aborts, then we need to read each entry in the undo log, and send the corresponding write to the memory system. This ensures that the changes made by the transaction are not visible to any other thread. Note that as compared to commits, aborts are more expensive in terms of time.

Important Point 16

Let us provide the main insight regarding flash clearing. The reader might want to go through the relevant background in Section 7.3 before reading this paragraph. If a cache line is 64 bytes, only one or two additional bits are used to store transactional state, and they may need to be flash cleared after a transaction commits or aborts. It is often a good idea to create a separate subarray to store the bits that need to be flash cleared. We need to support two kinds of accesses for this subarray. We need to read/write the bits, and flash-clear them. For reading and writing, we can use the same mechanism as the data array, where the decoder drives the corresponding word line to high, and then we read the value through the bit lines. For flash clearing, we need to ensure that all the cells in the entire subarray store a logical 0 after the operation is over. One solution is to enable all the word lines, set one bit line to a logical 0, and the other to a logical 1. By using this approach, we can write a logical 0 to all the memory cells in a single cycle. Another approach is to create a 2-ported memory, where we have two word lines. One word line can be used for regular access, and the other can be used for the purpose of flash clearing (writing a 0 to the cell). These approaches

have different trade-offs in terms of the complexity of the decoder and the overheads in creating an additional memory port.

Regardless of the design, simultaneously writing to an array of memory cells is difficult. We typically need a large amount of current to charge or discharge so many transistors. This places an unreasonable demand on the power grid. Hence, a lot of practical flash clearing systems [Miyaji, 1991, Rastegar, 1994] propose to divide the subarray into different continuous groups of memory cells. We clear them in stages. We first clear the bits in the first group, then after a given time delay we move to the next, and so on. This ensures that at no point of time, we place an unreasonable demand on the power grid of the chip. This does increase the latency of the entire operation; however, faults related to an excessive current draw do not happen.

Lazy Version Management

In lazy version management, we have a redo log that unlike an undo log stores data written by the transaction. All the variables in a transaction's write set have an entry in the redo log. Whenever, a transaction writes to a variable for the first time, it adds an entry for it in the redo log. Any subsequent read request made by the transaction needs to check for the variable in the redo log first. If the variable is present, then we treat its value as the current value of the transactional variable. If we do not find an entry, then we need to read the value of the variable from the regular memory system.

In this case, commits are more expensive than aborts. While committing a transaction, we need to write its entire redo log to memory. Moreover, all read requests in a transaction now have to be routed through the redo log. If they find their data in the redo log, then they need to use it. However, if they do not find their data, they need to read it from the regular memory system. The redo log basically acts like a cache. In the case of aborts, we simply need to discard the redo log; nothing else needs to be done.

The redo log per se can be stored as a software structure or as a separate hardware buffer. The good thing about a redo log is that it is more flexible, and allows us to support very large transactions. Since buffering the transactional state is an issue with eager version management, this approach is more scalable.

Way Point 13

- *The methods to manage concurrent transactions are collectively known as concurrency control mechanisms. There are two broad families of approaches: optimistic concurrency control and pessimistic concurrency control.*
  - *In optimistic concurrency control, we detect and recover from a conflict possibly after it has occurred. This means that we execute instructions after the conflicting accesses, and fix any resultant problems later.*
  - *In pessimistic concurrency control, whenever a conflict occurs, we immediately detect it and try to resolve it.*
- *There are two ways for detecting conflicts: eager and lazy. Eager conflict detection implies that we detect a conflict as soon as it occurs, as opposed to lazy conflict detection where we detect it much later.*

- *On similar lines, we have two kinds of version management: eager and lazy.*
  - *Eager version management implies that we write directly to the memory system. We maintain an undo log. We flush it if the transaction commits, and restore the state upon an abort. With this scheme commits are much faster than aborts. The main problem is maintaining the isolation property, where we need to ensure that other transactions are not able to read the temporary state of a transaction.*
  - *Lazy version management requires a redo log. While a transaction is active, all the writes are sent to the redo log. It acts as a temporary cache for the transaction, which the read operations need to check first. In this case, aborts are fast because we just need to discard the redo log; however, commits are slow because the entire contents of the redo log need to be written to the program's permanent state.*

9.7.2 Correctness Conditions

The same way we defined correctness conditions for parallel programs, we need to define some correctness conditions for transactions. There are a few widely used models for transactional correctness. Let us discuss a few of them. Note that in the following subsections, we only discuss correctness models for transactions. We do not discuss the interactions of transactions with non-transactional instructions. These are known as *mixed mode accesses*, and will be discussed later.

Serializability

This is a direct import from the world of databases with the same meaning. It states that a parallel execution with transactions should be equivalent to a serial execution with the same set of transactions. In other words, it should be possible to order the transactions in some sequence such that the results of both the executions are the same. In general, it is assumed that the transactions issued by the thread take effect in program order, hence, we shall use this as a necessary property in the definition of serializability. This is like sequential consistency at the level of transactions. Furthermore, the property of serializability does not specify the behavior of transactional accesses with respect to non-transactional accesses.

Strict Serializability

This is an extension of serializability, where we consider the real-time order as well. If transaction TS_A created by thread 1 completes before transaction TS_B (created by thread 2) starts, then the property of serializability does not say anything about how they should be ordered in the equivalent sequential order. TS_A can be ordered before TS_B or vice versa. However, strict serializability says that if TS_B begins after TS_A completes, then TS_B has to be ordered after TS_A in the sequence. If TS_A and TS_B are concurrent, which means that they overlap in time, then they can appear in any order in the equivalent sequence. For non-concurrent transactions, this property effectively enforces a real-time order on the transactions.

Opacity

The main problem with strict serializability can be seen in the following example (see Figure 9.60). Here, we read from two variables x and y. Initially, both of them are initialized to 0. The transaction by thread 2 sets both of them to 5. Now, assume that we have lazy conflict detection (at commit time), and eager version management. If we take a look at the transactions, we can quickly conclude that $t1$

should always be equal to $t2$. If the transaction of thread 2 executes first, then both x and y are equal to 5, otherwise both of them are equal to 0. It will never be the case that $t1 \neq t2$. Thus, thread 1 will never go into an infinite loop.

Listing 9.3: Thread 1

```
atomic {
    t1 = x;
    t2 = y;
    while (t1 != t2) {}
}
```

Listing 9.4: Thread 2

```
atomic {
    x = 5;
    y = 5;
}
```

Figure 9.60: A code snippet showing the need for opacity (adapted from [Harris et al., 2010])

This argument is correct for committed transactions because committed transactions need to follow all the ACID properties. However, we cannot say the same for aborted transactions that can read incorrect data, and then as a consequence get aborted. In this case, we use eager version management, which means that as soon as we effect a write, the value is visible to the rest of the transactions. They can access the variable; however, they may get aborted in the future. Here, thread 2 writes 5 to x. Then transaction 1 begins. It reads $x = 5$ and $y = 0$. In this case, thread 1 needs to get aborted. This will only happen when it reaches the end of the transaction. Sadly, before it reaches the end, we check if $t1 = t2$. This turns out to be false, and thus thread 1 goes into an infinite loop and never aborts. This behavior was not expected.

We observe this behavior because we did not define the correctness semantics for aborted transactions. We only defined them for committed transactions. Given that the transaction by thread 1 is aborting, we assumed that it need not follow any rules. This however lead to an infinite loop, and as far as the entire system is concerned, this execution is incorrect. We thus need to define a correctness model for aborted transactions as well. This model is known as *opacity*, which extends strict serializability by saying that it should be possible to order all transactions – committed, running or aborted – in a linear sequence. Every transaction Tx, committed or aborted, needs to see a consistent state, which is defined as the state produced by all the committed transactions ordered before Tx in the linear sequence. Furthermore, no transaction should be able to see the writes made by an aborted transaction. The execution in Figure 9.60 will not lead to an infinite loop if the TM system follows opacity.

Mixed Mode Accesses

Till now, we have only discussed correctness models for transactions. We have not included non-transactional accesses in our discussion. Let us now consider non-transactional accesses as well. Since we are considering both transactional and non-transactional accesses, let us refer to such accesses as *mixed mode accesses*.

Single Lock Atomicity

The simplest correctness model in this space is known as *Single Lock Atomicity* or SLA. We assume a hypothetical lock variable that has a global scope. Let us consider an execution to be valid if all the transactions *appear to* first acquire the hypothetical global lock, and then release it at commit/abort time. If an execution or a TM system follows this property, then we say that it satisfies single lock atomicity.

This model takes mixed mode accesses into account very well. We can borrow all the results from an equivalent lock based program, where we replace each transaction begin event with an acquire operation that acquires the global lock, and we replace the completion (abort/commit) of each transaction with a

lock release operation. For example, in a TM system with SLA we have a data race, if the equivalent lock based execution has a data race.

The main problem with SLA is that it does not allow disjoint access parallelism, and creates unnecessary dependences between all the transactions in the system. Now, all of them need to acquire the same global lock. This defeats the purpose of having a TM system.

Disjoint Lock Atomicity

The problem of an absence of disjoint access parallelism in SLA is readily solved by adopting another correctness criterion called *Disjoint Lock Atomicity* or DLA. Here, a transaction acquires all the locks for the variables that it accesses before hand. Then the transaction progresses. It finally releases all the locks when the transaction finishes. This model allows disjoint access parallelism, and we can realize many of the expected gains of transactional memory.

Even though DLA sounds very appealing, its biggest drawback is that we need an exact knowledge of the locks that a transaction needs. Later transactional models (see Section 9.7.3) have relaxed this requirement to make the DLA model more practical. They require the transaction to acquire locks for all the variables that it accesses just before the first access to each variable. We thus need not know which variables a transaction is going to access a priori.

Transactional Sequential Consistency

Transactional sequential consistency (TSC) is defined as an extension of sequential consistency for TM systems. It is defined on the same lines as opacity and traditional sequential consistency. It says that we can order all transactions (committed or aborted) and all non-transactional instructions in a linear sequence. In this sequence, all the instructions including the ones within transactions appear in program order.

We have the notion of transactional data race freedom (TDRF), which prohibits races between non-transactional accesses, and races between transactional and non-transactional accesses. On similar lines, we can prove that TDRF programs obey TSC.

9.7.3 Software Transactional Memory

Software Transactional Memory (STM) is a popular paradigm for implementing transactional memory systems. An STM is easy to use, and it requires changes to just the compiler or the runtime. Unlike hardware transactional memory, it does not suffer from limitations of space. We shall describe two commonly used STM algorithms in this section. They are at different points in the design space of transactional memory technologies.

First, we need to augment each variable used in transactions with some additional data called *metadata*. This metadata is used to track different versions of a variable, check for conflicts, and effect commits and aborts. Different algorithms use different kinds of metadata. In addition, each transaction maintains a read set and a write set. The read set contains the set of variables read by the transaction, and the write set contains the set of variables written by the transaction.

Finally, the compiler or the runtime convert every read or write operation into equivalent $readTX$ and $writeTX$ operations. In addition to performing regular reads and writes, these operations execute additional code to check for conflicts, and also buffer either speculative data, or data that has been overwritten. Let us look at two popular designs in this category.

Bartok STM

The Bartok STM [Harris et al., 2006] uses optimistic concurrency control for reads, with eager version management (an undo log), and lazy conflict detection.

Figure 9.61: Structure of a transactional variable

Every transactional variable has three fields (see Figure 9.61): value, version, and lock. The *value* field (as the name suggests) is the value of the variable. The version is a monotonically increasing integer that indicates the version of the variable. Every time we write to the variable, the version number is incremented. Finally, the *lock* field is a 1-bit value that indicates if the variable is locked or not.

Read Operation

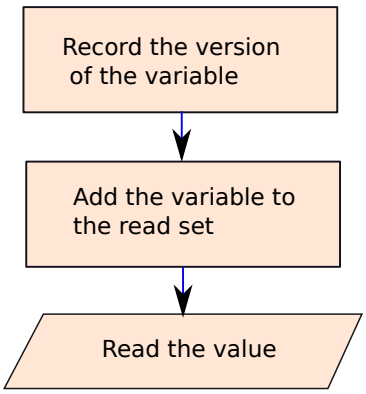

Figure 9.62: A read operation in the Bartok STM

A basic read operation is very simple as shown in Figure 9.62. We first record the version of the variable, then we add it to the read set of the transaction, and finally we read the variable and return its value. The main reason for recording the version of the variable is to use this information to detect a conflict later. If it is found out that we read an outdated version, then the transaction needs to be aborted. We shall see later that the version of a variable is incremented if a write to that variable commits.

Write Operation

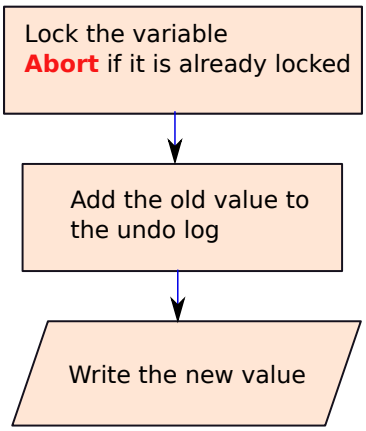

Figure 9.63: A write operation in the Bartok STM

In the write operation, we first try to lock the variable. This is to ensure that no other thread modifies the variable during the transaction. If the variable is already locked by another transaction, then we abort the transaction. In other words, this means that the current transaction cannot proceed. If we are successful in getting the lock, then we add the old (previous) value of the variable to the undo log. The *undo log* in this case is a region in software that stores the old values of variables. Once, we have added the value to the undo log, we proceed to effect the write. In this protocol, both the read and write operations are simple. Let us now look at the commit operation. Figure 9.63 shows the flow of actions.

Commit Operation

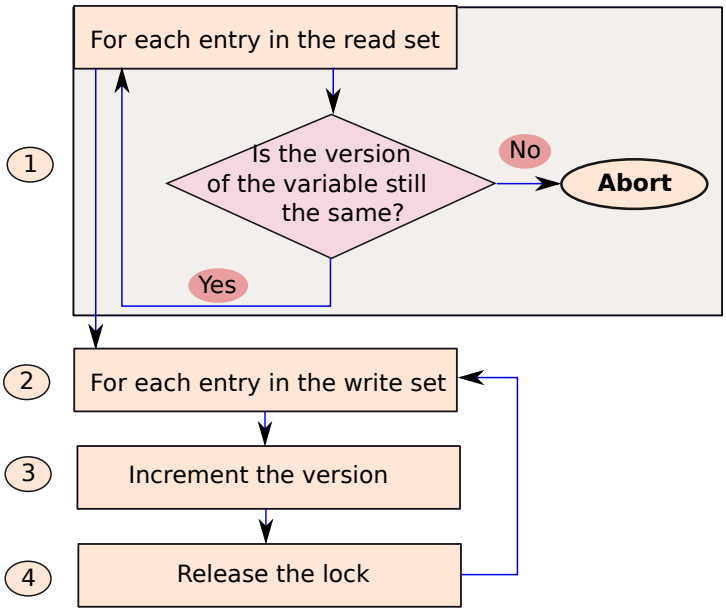

Figure 9.64: The commit operation in the Bartok STM (the numbers indicate the sequence of actions)

We commit a transaction when we finish executing the last instruction in the transaction. The commit protocol is shown in Figure 9.64. We have separate actions for the read set and the write set. For each entry in the read set, the protocol is as follows. For each variable, we compare its recorded version (at the time it was read by the transaction for the first time), and the current version. If the versions are not the same, then we can conclude that there was an intervening write by another transaction. Thus, the current transaction needs to abort because in this case the two transactions are conflicting. After we abort the transaction, we release all the locks.

For each entry in the write set, we first increment the version, and then release the lock. This ensures that all other transactions see the results of this transaction, and also perceive the fact that the variable has been updated. Once we have committed a transaction, we can discard its undo log. Let us now understand why this protocol works.

Consider a read operation for variable x. Between the time that it is read for the first time in the transaction, and when we commit the transaction, we are sure that no intervening write has committed. A transaction that writes to x is thus ordered after the current transaction. Hence, a read-write conflict is handled correctly. Now consider write operations. We need to lock a variable during the lifetime of its use within a transaction. We lock it the first time that we use it, and keep it locked till we are ready to commit the transaction. This ensures that no other transaction can write to the variable. Any transaction that will write to the variable has to wait till the current transaction is over. This ensures that we do not have write-write conflicts in our system.

Let us now consider the pros and cons. This approach is simple, and read operations simply need to record the version of the variable. However, write operations are expensive. It is necessary to lock the variables, and this increases their delay. Since this method uses lock and unlock operations, the performance is dependent on how many variables within a transaction need to be locked, and how long it takes to acquire a lock.

Since this protocol uses eager version management, commits are fast because nothing needs to be written to memory. The final state has already been written to memory. However, aborts are more expensive because we need to restore the state of all the variables that have been written to. This is done by reading the undo log, and replacing the contents of each entry with the value stored in the undo log.

From the point of view of correctness, this protocol provides a strong semantics for transactions in the sense it ensures that all the transactions are serializable. However, it does not provide opacity, which also mandates that aborted transactions see a consistent state.

TL2 STM

Let us now look at another STM solution that works very differently, yet provides opacity. It is known as the TL2 STM [Dice et al., 2006]. Unlike the Bartok STM, it uses lazy version management, which means that it requires a redo log. In this transaction memory protocol, we have a monotonically increasing atomic global counter that provides a timestamp to every invoking process. Every time a transaction Tx starts, it reads and increments the global counter. The timestamp provided by the global counter is stored as $Tx.rv$ (field called rv (read version) in the transaction). In addition, the metadata corresponding to each transactional variable contains two additional fields: a timestamp and a lock.

Read Operations

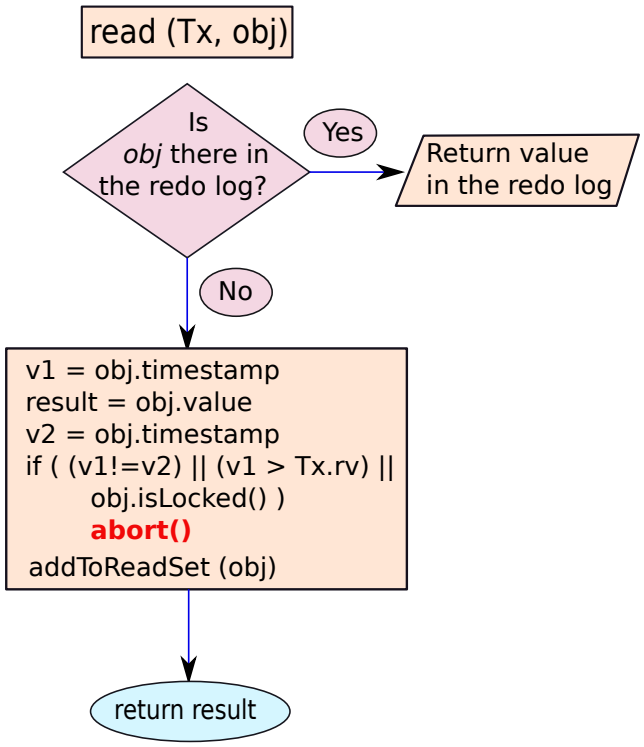

Figure 9.65: The read operation in the TL2 STM (adapted from [Harris et al., 2006, Harris et al., 2010])

Figure 9.65 shows the flow of the read operation. Since we have a redo log, whenever we read a value, we need to check in the redo log first. The redo log is a software structure that keeps the values of transactional variables till the transaction is over. If the variable is present in the redo log, then we return the value, otherwise we need to follow a complex protocol.

We first record the timestamp of the variable in $v1$. Then we read the value of the object, and then we read the timestamp of the variable once again, and store it in $v2$. Then we check a couple of conditions. If any one of them is true, we need to abort the transaction. The reasons are as follows.

[$v1 \neq v2$] This means that the variable has possibly changed between the time that it was read and the time that we are checking the timestamp for the second time. This algorithm is known as an *atomic snapshot*. The reason we need to do this is as follows. We are reading the value of the variable and the timestamp at the same time. It is possible that we read the timestamp first and then the variable changes. Then both the pieces of information will be out of sync. There is thus a need to read the timestamp once again and verify that it is the same. We will be sure that we have atomically collected a snapshot of both the variable and its timestamp. This is a standard technique that is used to read an object that spans multiple memory words. We read the timestamp twice – once before reading the variable and once after reading the variable.

[$obj.isLocked()$] If the variable is locked by some other thread, then this variable is in the process of getting updated. Its value cannot be read at the moment. Thus, the current transaction needs to abort.

[$v1 > Tx.rv$] This means that some other transaction has incremented the timestamp of the variable, after the current transaction began. We cannot guarantee the isolation of transactions, and thus the current transaction has to abort.

Note that in this case all the checks are being done at the time of reading. We are ensuring that a value that is being read is safe to read. We then add the variable that was read to the read set and proceed.

Write Operations

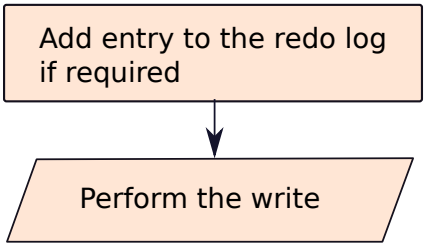

Figure 9.66: The write operation in the TL2 STM

A write operation is far simpler (see Figure 9.66). We just add an entry into the redo log if it is not already there, and we go ahead and perform the write. Note that in this case, the value that is written is sent to the redo log. Writes are made permanent only while committing.

Commit Operation

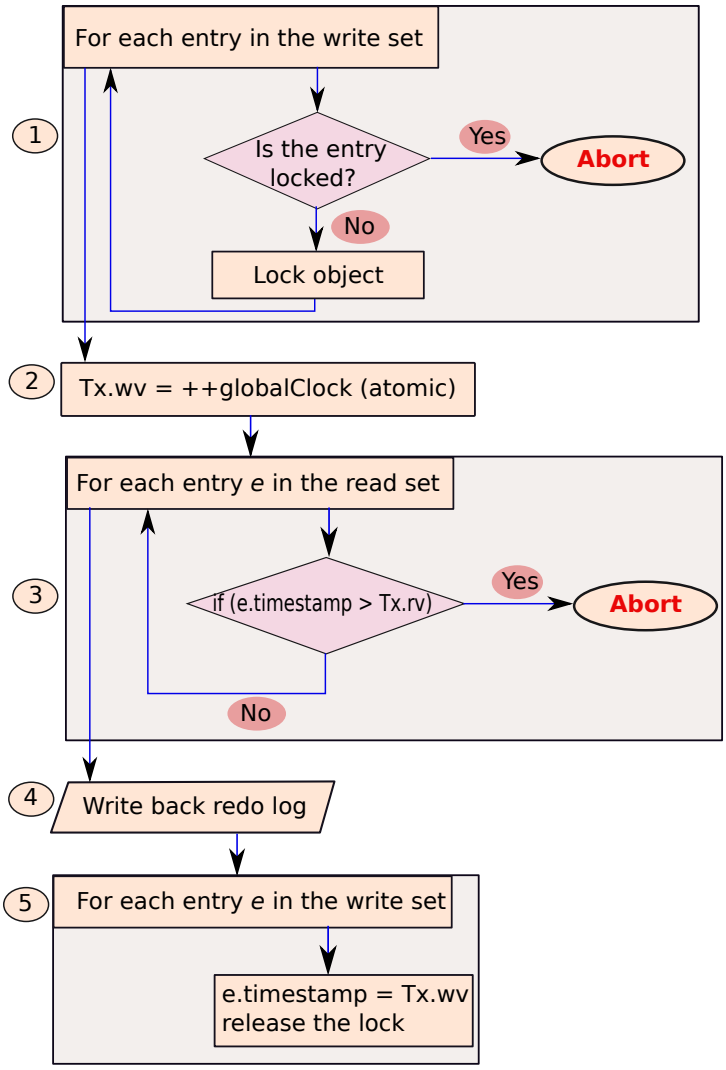

Figure 9.67: The commit operation in the TL2 STM (adapted from [Harris et al., 2010, Harris et al., 2006])

Figure 9.67 shows the flow of actions while performing a commit. For each entry in the write set, we lock the variable. If we are not able to lock any variable, then we need to release all the locks that we have obtained, and abort.

Now, assuming that we have gotten all the locks, we increment the global timer and get a new timestamp for the current transaction, which is stored in the variable $Tx.wv$ (write version). Next, we validate the read set. Note that in this protocol reads are expensive. We did a round of validations while reading a variable for the first time, and we need to do another round of validations at the time of validating the read set.

As shown in Figure 9.67, we compare the timestamp of each variable with the read timestamp of the transaction, $Tx.rv$. Recall that we had collected the read timestamp when the transaction began. This comparison checks if the variable has been updated after the current transaction began. If it has, then we need to abort the current transaction.

At this point, the read set and the write set have been validated. We can thus proceed with performing the writes. We read all the entries in the write set, get their values from the redo log, and write them to memory. These writes make the transaction visible. Once we are done with the writes, the redo log

can be discarded.

Finally, we set each variable's timestamp to $Tx.wv$, and then we unlock all the variables in the write set. This finishes the commit process. We need to note a couple of subtle points in this algorithm.

1. With a redo log, commits are more expensive than aborts. If we need to abort the transaction, we just need to release all the locks and discard the redo log. Commits in comparison are more expensive.

2. We use two timestamps per transaction: $Tx.rv$ and $Tx.wv$. $Tx.rv$ is set at the beginning of a transaction by reading the value of the global counter, whereas $Tx.wv$ is set at the time of committing the transaction. $Tx.wv \geq Tx.rv + 1$.

3. We first write the variables to permanent state, and then we update their timestamps. This ensures that if another transaction sees an updated timestamp, it is certain that the changes have been made to the permanent state.

4. As compared to Bartok STM, we do not hold locks for very long. They are only held for the duration of the commit operation. This is expected to be a short duration since the commit operations are a part of the transaction manager library and large delays are not possible by design.

9.7.4 Hardware Transactional Memory

Software transactional memory (STM) systems have numerous shortcomings. They essentially convert reads and writes into function calls. For every read and write in a transaction, it is necessary to call a function that records the version of the variable that is read/written, makes changes to the undo/redo log, and acquires a lock. Furthermore, at the end of a transaction, it is necessary to look at every single read and write in the read and write sets and take appropriate action. This can involve releasing the locks, comparing versions, and aborting the transaction (if necessary). Coming to correctness, even in models that provide opacity, they do not guarantee safety against data races when one of the accesses is outside the scope of a transaction. It is thus necessary to create support for supporting transactional operations in hardware.

Hardware transactional memory (HTM) has numerous advantages over STMs. Individual operations such as reads, writes, transaction begin and end events are much faster. In addition, maintaining undo and redo logs is done at the level of hardware, which can be done very efficiently. Along with performance advantages, hardware transactional memory requires little software support other than additional instructions to mark the beginning and end of transactions, and methods to indicate if a transaction has aborted or committed.

Coming to disadvantages, HTM systems have plenty of them. As is common in hardware based implementations, such augmentations increase the complexity of hardware and increase power consumption. Furthermore, hardware has more resource constraints. For example, if we wish to maintain undo or redo logs in hardware then this limits the size of a transaction. If a transaction requires more storage for storing values, then we need to either abort the transaction or create a complex mechanism to dynamically allocate more memory to the transaction from the regular memory space.

Let us appreciate these trade-offs by looking at the design of a HTM, which is inspired by LogTM [Moore et al., 2006] [4].

ISA Support

We need to add some extra instructions to the ISA. These instructions mark the beginning and end of transactions. Most versions of hardware transactional memory typically add three instructions: *begin*,

[4] The protocol that we describe is not exactly similar. Some simplifications and modifications have been made.

commit, and *abort*. An *abort* instruction is required to enable the software to automatically kill a transaction if a special circumstance arises. By default, the compiler or programmer place a *commit* instruction at the end of a transaction.

If we have nested transactions (transaction within a transaction), then the *begin* instruction increments the nesting level, and the *abort* and *commit* instructions decrement the nesting level. Transactions typically contain simple processor instructions that only make changes to memory and the registers. Most implementations of transactional memory do not allow transactions to make system calls or write to I/O devices.

Version Management

In any HTM protocol, we have a choice between eager and lazy version management. From the point of view of performance, using eager version management with an undo log is better, particularly if we have large transactions. In this case, values can be read and written directly to memory. We do not have to maintain a separate data structure to hold the values of transactional variables.

In our HTM, we shall use eager version management. Each thread creates an undo log in its virtual memory space. This log is stored in the physical memory space and can be cached. The algorithm for reads and writes is as follows. Whenever a transaction begins, the core sets a bit and remembers the fact that it is in a transaction. Till the transaction ends, we need to keep track of the read set and the write set. This is required to detect conflicts.

To help in this process our HTM adds two bits to every L1 cache line: R (read) and W (write). The R bit is set when we read a word in the line. When we write to a word in the line, we set the W bit. We need not set it all the time; we can set it only once at the time of the first write access. At this point of time, it is also necessary to write the previous value to the undo log, which is a dedicated memory region in the process's virtual address space. For subsequent writes to the same block, it is not necessary to modify the undo log. Once the transaction is over, there are fast mechanisms to quickly clear all the R and W bits within a few cycles. These are known as *flash clearing* mechanisms in caches [Miyaji, 1991, Rastegar, 1994] (see Point 16).

The main advantage of the R and W bits is that they identify the variables that have been read and written in a transaction. This information can then be used to detect conflicts. They implicitly represent the read and write sets.

Conflict Detection

The main advantage of using hardware is eager conflict detection. Unlike software based methods, where we need to perform elaborate checks, a hardware based conflict detection scheme can leverage the coherence protocol. Eager conflict detection saves a lot of wasted work. Secondly, since all processors support coherence, a minor modification to the coherence protocol to support transactions does not represent a significant overhead.

Whenever a given word is not there with a core, it sends a request to the directory asking for either read access or write access. If it is a read request, the directory forwards it to the cache that has a copy of the block. If it is a write request, then the directory needs to invalidate all the copies of the block that are there with other sister caches. In both cases, it needs to send a message to a set of sister caches, indicating that one of their blocks needs to be read or written by another cache.

This is where we can detect a conflict. For the subsequent discussion, let us assume a system with coherent L1 caches; it forwards all the directory messages to the cores, which forward them to their attached L1 caches after some processing. There are two kinds of replies that a core can send to a directory: *ack* and *nack*. It sends an acknowledgement (*ack*) if the access does not conflict with its read set or write set; otherwise, if there is a conflict, then it sends a *nack* message. This lets the directory know that a conflict has occurred; the directory then forwards this message to the requesting core. Once a conflict is detected it needs to be resolved, which means that one of the transactions involved in the conflict needs to either wait or get aborted.

The main problem with such kind of conflict detection is that if a cache evicts a block, and if the directory also removes its corresponding entry, then we will have no record of the fact that a given block is in the read or write set of a given core. This means that if a cache evicts a block that is a part of the transactional state of the core, the directory still cannot remove it completely. It can be removed from the list of sharers; nevertheless, its state still needs to be kept in the entry of the directory.

There are two cases: the block was in the M state or in the S state. When core C replaces a block that was in the M state, its corresponding entry in the directory transitions to the state $M@C$ (referred to as a *sticky* state). For example, if core 2 replaced a block, then the state is set to $M@2$. In addition, C sets its overflow bit – assume that each core has a dedicated *overflow* bit, which is initialized to 0, and reset to 0 when a transaction ends (commit or abort). The state $M@C$ means that currently there are no sharers for this block; C does not have a copy of it in its cache, even though this block is in its write set. When another core requests for the block, the directory forwards the request to core C with its current state ($M@C$). C infers that this block must be in its write set. If the transaction is still going on, then there is a conflict, otherwise core C can return an acknowledgement (refer to Figure 9.68).

Now, consider the second case: the block in core C was in the S state. Depending upon the protocol, we can either have silent evictions (no messages sent) or the core might send a message to the directory. Consider the more difficult case, where the eviction is silent. In this case, the directory has no record of the fact that C is no more a sharer. The next time it gets a write miss request from another core, it forwards the request to C. This is where a conflict can be detected (similar to the earlier case with writes).

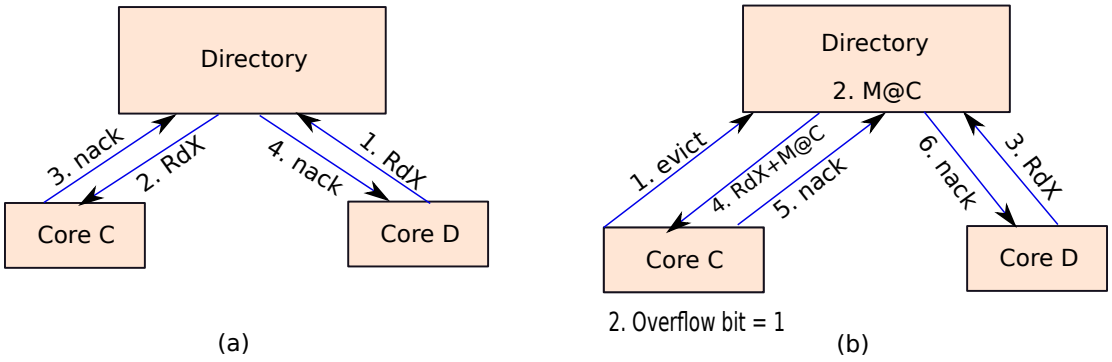

Figure 9.68: (a) Detection of a conflict (block present in the cache), (b) Detection of a conflict using the overlap and $M@C$ bits.

Subtle Correctness Issues

Let us conclude with looking at subtle correctness issues when it comes to evictions. If there is an eviction, then we need to send the block to the lower level. This means that we are also polluting the lower levels of the memory hierarchy with data that might belong to an aborted transaction.

Read and Write Sets

Let us first consider *what happens to the read and write sets* when a block is sent to the L2 cache from the L1 cache. In this case, we are maintaining the read/write sets implicitly using the R and W bits. If the block is evicted, this information is also gone.

However, we can avoid correctness issues by using the sticky states in the directory. They ensure that processing another access (transactional or non-transactional) will not lead to errors. If the other access is transactional, one of the transactions needs to abort because C will send a *nack* message. In this case, either the transaction on C will abort, or it will continue to run (other transaction will abort). In the

former case, the read and write sets need to be discarded anyway, and in the latter case, the status quo will continue. Now, if the other access was non-transactional, and we do not want it to wait, then the transaction running on core C needs to abort. This is because that other access cannot be rolled back because it is not a part of a transaction. The read and write sets will be discarded and this is the correct behavior.

Contents of Evicted Blocks

The other issue that we need to account for is the contents of evicted blocks that are possibly written to the lower levels of memory. If a transaction is active, then no other transactions or non-transactional reads/writes can make conflicting accesses to the locations in its read and write sets.

There are two cases here: the original transaction commits or aborts. If the original transaction commits, then there is no issue. However, if it aborts, then we may end up with incorrect data populating the L2 cache. Fortunately, this will not cause a problem because we *need to write back the contents of the undo log*. Consider a block b that was evicted by core C, and this block was written to the L2 cache. If the transaction aborts, then the old contents of b will be written to the L1 cache, and thus the correct state of the memory system will be restored. Note that if L1 contains a block in the modified state (because of a write from the undo log), then the contents in the L2 cache do not matter. It is anyway assumed to have a stale copy of the data.

Commits and Aborts

With eager version management, commits are always easy. We need to flash clear all the R and W bits, reset the overflow bit of the core, and clear the undo log. In this case, an additional action that needs to be taken is that we need to ensure that all the sticky states created in the directory because of the committed transaction are cleared. One easy option is to send a message to the directory with the core id, C. The directory can then walk through all the entries whose state is $M@C$, and clear their states. If the write set is very small, we can send messages for all the blocks in the write set as well.

If a transaction aborts, we need to restore each entry stored in the undo log. The time taken for this step is proportional to the size of the undo log. After restoring the memory state, we need to flash clear all the RW bits, and reset the rest of the states as we had done in the case of committing a transaction.

9.8 Summary and Further Reading

9.8.1 Summary

Summary 8

1. *There are two major paradigms in parallel programming: shared memory and message passing.*
   (a) *In the shared memory paradigm, we assume that all the threads share the memory space and communicate via reading and writing variables.*
   (b) *In the message passing paradigm, threads communicate explicitly by sending messages to each other.*
   (c) *The shared memory paradigm is typically used in strongly coupled systems such as modern multicore processors, whereas message passing is used in loosely coupled systems such as cluster computers.*

2. *The speedup with parallel execution as a function of the number of computing units, and the sequential portion of the benchmark is governed by the Amdahl's law.*

$$Speedup = \frac{1}{f_{seq} + \frac{1-f_{seq}}{P}}$$

Here, f_{seq} is the fraction of the execution that is sequential, and P is the number of processors.

3. *The Amdahl's law assumes that the size of the workload remains fixed as we scale the number of processing units. This is seldom true. The Gustafson-Barsis's law fixes this problem, and assumes that the parallel portion of the work scales with the number of processing units. The net speedup is thus as follows:*

$$Speedup = f_{seq} + (1 - f_{seq})P$$

4. *The Flynn's taxonomy defines the spectrum of multiprocessing systems: SISD (uniprocessor), SIMD (vector processor), MISD (redundant processing units in mission critical systems), and MIMD (multicores). MIMD processors can further be divided into two types: SPMD (master-slave architecture) and MPMD (regular multithreaded programs).*

5. *Hardware multithreading is a design paradigm where we share the pipeline between multiple concurrently running threads. Each thread has its PC, architectural registers, and rename table. The rest of the units are partitioned between the threads.*

6. *A typical multicore processor contains multiple processing cores that use the shared memory paradigm to communicate with each other. In such a system, having a single shared cache is not efficient in terms of performance, hence we need to have a distributed cache.*

7. *If a distributed cache follows the properties of coherence, then it appears to the program as a single shared cache. A distributed cache has a low access time and can support many parallel accesses by different cores.*

8. *A key correctness property of a memory system is PLSC (per location sequential consistency). This means that all the accesses to a single location can be laid out in a sequence such that each access is* legal *– every read gets the value of the latest write. PLSC needs to hold even in systems with non-atomic writes.*

9. *There are two fundamental axioms of cache coherence that naturally arise out of PLSC and the fact that in practical systems writes are never lost.*

   Write Serialization Axiom *A write to the same location is seen in the same order by all the threads.*

   Write Propagation Axiom *A write is eventually seen by all the threads.*

10. *The behavior of a memory system for multiple locations is governed by the memory model (or memory consistency model).*

11. *Sequential consistency (SC) is the gold standard for memory models. An execution is said to be in SC, if the memory accesses made by all the threads, can be put in some sequential order subject to the fact that in this sequential order the accesses of each thread appear in program order, and each read gets the value of the latest write.*

12. *SC forbids most optimizations such as write buffers, LSQs that send reads to the cache before earlier writes, complex NoCs that reorder messages, MSHRs, and non-blocking caches.*

13. *Hence, in practice, most memory consistency models relax the program order constraint because of performance issues. Many modern models such as those provided by IBM and ARM also allow non-atomic writes.*

14. *The standard theoretical tool to model executions is the method of execution witnesses. In an execution witness, we have four kinds of edges: a subset of program order edges (po), write $\rightarrow$ read dependence edges (rf: rfe and rfi), write serialization edges (ws), and read $\rightarrow$ write edges (fr). ws and fr edges are a direct consequence of PLSC, and are present in almost all systems. However, the po and rf edges are relaxed (not present in the execution witness) to different degrees in different memory models. In an execution witness, we add all the edges corresponding to a memory model, and if there are no cycles, then it means that the execution is consistent with the memory model.*

15. *We also need to obey uniprocessor access constraints such that single-threaded code executes correctly on a multiprocessor machine and PLSC is not violated.*

16. *Most systems prohibit thin-air reads. This means that some data and control dependence relations need to be respected by the memory model.*

17. *To implement coherence we need a cache coherence protocol. If we have a few cores, then we prefer snoopy protocols, where all the cores are connected with a single bus. Otherwise, we prefer the directory protocol, where the* directory *is a dedicated structure that is reachable via the NoC.*

18. *The two most common snoopy cache coherence protocols are the Write-Update and Write-Invalidate protocols.*

    (a) *In the Write-Update protocol, we broadcast every write to the rest of the sisters caches. Even though we broadcast writes very quickly and eagerly, this protocol has a large overhead due to the frequent write messages.*

    (b) *The Write-Invalidate protocol solves this problem by broadcasting messages to the rest of the sister caches only when there is a write miss.*

19. *We typically use the $MESI$ protocol to implement the Write-Invalidate protocol. Each cache line has four states: Modified (M), Exclusive (E), Shared (S), and Invalid (I). In the* Shared *state, the cache can only read the block, in the* Exclusive *state we are sure that no other sister cache has a copy of the block (read-only access), and in the* Modified *state the cache is allowed to both read and write to the block. These protocols have elaborate state transition diagrams that determine the rules for transitioning between the states. We can additionally add an O (Owner) state that designates a given cache as the owner of a block – it supplies a copy of the block if there is a remote request.*

20. *In the directory protocol, we typically have a few centralized directory structures that maintain the list of sharers for each cache block. Whenever there is a read miss, a message is sent to the directory, it adds the new cache to the list of sharers, and asks one of the sharers to send a copy of the block to the new cache. If there is a write miss, then the directory sends an invalidate message to all the sharers, and ensures that a copy of the block is sent to the cache that wishes to write to it.*

21. *Atomic instructions that are used to implement locks and critical sections, are implemented using extensions of the coherence protocol. Different atomic instructions are powerful to different degrees; this is captured by the* consensus number.

22. *To implement different memory models, we need to explicitly enforce different orderings. This often requires sending acknowledgements for write completion and ensuring that the ordering of regular instructions with respect to synchronization instructions is respected.*

23. *A* data race *is defined as a conflicting pair of accesses of a regular variable by two concurrent requests across threads. When two requests access the same variable, where at least one of them is a write, they are said to be* conflicting. *Two requests are said to be* concurrent*, when there is no path between them in the execution witness that contains a synchronization edge (edge between two synch operations).*

24. *Data-race-freedom implies SC. However, it is possible for an SC execution to have a data race. If we enclose all accesses to shared variables in critical sections and consequently disallow concurrent accesses, we can prevent data races. Such programs are said to be* properly synchronized.

25. *If a program has a data race on a machine that uses a non-SC memory model, then we can construct an execution of the program that has a data race and is in SC.*

26. *There are two common approaches for detecting data races: the lock set algorithm, and the algorithm based on vector clocks.*

27. *Traditional programming that uses critical sections is difficult for most programmers, and many desire simpler abstractions. Hence, the paradigm of transactional memory was developed, where all that a programmer needs to do is mark a block of code as* atomic*. The runtime ensures that the block runs atomically, and it is not possible for any other thread to see a partial state (state in the middle of an atomic block's execution). Such atomic blocks are known as* transactions*, and such a system is known as a* transactional memory system.

28. *There are two kinds of transactional memory systems: STMs (in software) and HTMs (in hardware).*

    (a) *Transactions typically have a* begin *and* end *operation.*

    (b) *Every transaction has a read set and a write set – the set of variables read and written respectively. Two transactions T_i and T_j are said to conflict if either $R_i \cap W_j \neq \phi$, or $R_j \cap W_i \neq \phi$, or $W_i \cap W_j \neq \phi$.*

    (c) *If a transaction executes correctly without any conflicts (concurrent conflicting accesses), then it can commit (make its changes visible to the rest of the threads), else it needs to abort (none of the changes made by it are visible).*

    (d) *We can either detect conflicts eagerly or lazily (when the transaction ends).*

    (e) *Every transaction needs to temporarily buffer its state till it commits. There are two approaches in this space. Either it can eagerly write to the memory system, and rollback changes later if there is an abort (using an undo log). Conversely, it can adopt a lazy approach and buffer its writes in a redo log during its execution. The entries in the redo log can then be written to the memory system (permanent state), if the transaction commits.*

29. *STM systems instrument the transaction begin, end, commit, and abort operations to track the version of each variable, perform book keeping, and in some cases lock a few variables. When the transaction ends, they check if there have been any conflicting accesses during the lifetime of the transaction, and if there have been, then one of the conflicting transactions needs to abort. Otherwise, the changes are made permanent (committed). We discussed two STMs in this chapter: the TL2 and Bartok STMs.*

30. *Hardware transactional memory (HTM) systems modify the coherence protocol to track conflicting accesses to variables within the scope of transactions, and use this information to abort or commit transactions.*

9.8.2 Further Reading

Readers should start this chapter by first learning how to write parallel programs. They can refer to the book by Quinn [Quinn, 2017]. If they would like to get a better understanding of the material on the theory of memory models, then it is advisable that they read the first few chapters of the book by Herlihy and Shavit [Herlihy and Shavit, 2012]. This will teach them all about sequential and parallel executions, legal sequences, lock-free algorithms, and consensus numbers.

For cache coherence readers can start with a survey of implementations of cache coherence protocols by Stenstrom [Stenstrom, 1990] and then proceed to read the book by Sorin [Sorin et al., 2011]. For an early implementation of the directory protocol in the Stanford Dash multiprocessor, the paper by Lenoski et al. [Lenoski et al., 1990] is the most definitive reference. Modern implementations are described in the references [Abts et al., 2003, Vantrease et al., 2011].

For memory consistency models, readers should first read the tutorial by Adve and Gharachorloo [Adve and Gharachorloo, 1996] followed by the theses of the authors [Adve, 1993, Gharachorloo, 1995]. These references are at a fairly high level, to get a deeper theoretical understanding, it is necessary to read papers with more formal methods such as the papers by Alglave [Alglave, 2012], and Wickerson et al. [Wickerson et al., 2017]. For a micro-architectural perspective, we recommend the papers by Arvind et al. [Arvind and Maessen, 2006] and Lustig et al. [Lustig et al., 2014]. An important work in this space that generates tests for different memory models is the work by Mador-Haim et al. [Mador-Haim et al., 2011].

Much of the theory of DRF memory models was developed by Adve in her thesis [Adve, 1993]. Subsequently, two papers discussed lockset based data race detection in software [Savage et al., 1997] and hardware [Zhou et al., 2007] respectively. Readers can refer to Prvulovic et al. [Prvulovic, 2006] for a method to detect races in hardware using simple timestamps.

For transactional memory, the book by Harris, Larus, and Rajwar [Harris et al., 2010] is a comprehensive reference. For important theoretical results refer to the work by Guerraoui [Guerraoui and Kapalka, 2008, Guerraoui and Kapałka, 2010].

Exercises

Ex. 1 — Write a shared memory program to perform merge sort in parallel.

Ex. 2 — What are the pros and cons of the shared memory and message passing schemes.

Ex. 3 — Why is it often better to use the Gustafson-Barsis's law in place of the Amdahl's law?

Ex. 4 — Why do we write the block back to the lower level on an $M \rightarrow S$ transition?

Ex. 5 — What is false sharing? How can we avoid it?

Ex. 6 — What are the advantages of the directory protocol over a snoopy protocol.

Ex. 7 — Consider a regular MESI based directory protocol, where if a line is evicted, we do not inform the directory. What kind of problems will this cause? How do we fix them?

* **Ex. 8** — In the MOESI protocol, we may have a situation where a block does not have an owner. This is because we do not have a mechanism for transferring the ownership. Propose a solution to this problem that has the notion of ownership transfer.

** **Ex. 9** — We need to create a new instruction called MCAS (multi-word CAS). Its pseudocode is as follows.

```
boolean MCAS (int* addresses[N], int oldValues[N], int newValues[N]) {
      int i, flag = true;
      for (i=0; i < N; i++) {
            if (*addresses[i] != oldValues[i]) {
                /* old value does not match */
                flag = false;
                break ;
            }
      }

      /* at least one of the old values did not match */
      if (flag == false) return false;

      for(i=0; i<N; i++)   /* set all the new values */
            *addresses[i] = newValues[i];
      return true;
}
```

1. Provide a hardware implementation of MCAS that makes it appear to execute atomically. What changes do we need to make to the ISA, the pipeline, and the memory system. Note that we have to introduce a simple RISC instruction called MCAS. How do you give it so many arguments?
2. Given two variables stored in different locations in memory, we need to read an atomic snapshot where the snapshot contains a pair of values (one for each variable) that were present at the same point of time. We cannot use normal reads and writes (values might change in the middle). How can we use MCAS to do this? [Hint: Use timestamps]
3. Use MCAS to implement lock and unlock functions. Show the code.

* **Ex. 10** — Assume a cache coherent multiprocessor system. The system issues a lot of I/O requests. Most of the I/O requests perform DMA (Direct Memory Access) and directly write to main memory. It is possible that the I/O requests might overwrite some data that is already present in the caches. In this case, we need to extend the cache coherence protocol that also takes I/O accesses into account. Propose one such protocol.

Ex. 11 — Does PLSC guarantee SC? Does SC guarantee PLSC?

* **Ex. 12** — How does the method of execution witnesses provide an illusion of sequential execution?

* **Ex. 13** — You are given a machine with many cores. You don't know anything about the memory model that it follows. You only know that the rfe order is *global*. You are allowed to write parallel

programs, give them as input to the machine, and note the outcomes. If you run the program for let's say a million times, it is **guaranteed** that you will see all the possible outcomes that the memory model allows.
Write four programs to find if each of these four orders hold: $W \rightarrow R$, $W \rightarrow W$, $R \rightarrow W$, $R \rightarrow R$ (R means read, W means write). Prove that your approach will work using the method of execution witnesses. Try to minimize the number of instructions.

** **Ex. 14** — Consider the following relations between two loads, L and L', in a multiprocessor system.

$$\begin{aligned} & loc(L) = loc(L') \quad (same\ location) \\ & \wedge (A, L) \in ghb \\ & \wedge (A, L') \in ghb \\ & \wedge (source(L), B) \in ghb \\ & \wedge (source(L'), B) \in ghb \end{aligned}$$

$loc(L)$ refers to the memory address of L. Consider A and B to be two other memory accesses. ghb is the global happens before order. $(X, Y) \in ghb$ means that X needs to happen before Y. $source(L)$ refers to the store that produces the value for load L ($source(L) \xrightarrow{rf} L$).
Does $(A, B) \in ghb$ hold for all standard memory models, or only for some?

** **Ex. 15** — What changes should be made to the pipeline and the memory system to ensure that thin air reads do not happen with value prediction?

** **Ex. 16** — Consider the following code for the Peterson lock with two threads. The threads are numbered 0 and 1 respectively. For a thread, we assume that the function $getTid()$ returns the id of the thread. It can either be 0 or 1. If $(getTid() = t)$, then t is the id of the current thread, and $(1 - t)$ is the id of the other thread. *turn* and *interested* are global variables. Rest of the variables are local.

```
void lock(){
     int tid = getTid();
     int other = 1 - tid;

     interested[tid] = true;
     turn = tid;

     while ( (interested[other] == true) && (turn == tid))
        { /* keep looping */ }

     /* lock acquired */
}

void unlock(){
     int tid = getTid();
     interested[tid] = false;
}
```

1. Prove that this algorithm is correct in a sequentially consistent system.
2. Will this algorithm work in a system with weak consistency? Explain your answer.
3. Consider the TSO memory model that Intel uses. It has atomic writes, and the only ordering that is relaxed is the $Write \rightarrow Read$ ordering. Where do we need to add fence instructions? Explain the correctness of the code with fences.

For all the three problems preferably use execution witnesses.

** **Ex. 17** — In any execution witness with a cycle, and different addresses, is it possible to have a single *po* edge? Justify your answer.

** **Ex. 18** — Is it true that a memory model = atomicity + ordering? Prove your answer.

Ex. 19 — Consider the *RCpc* memory model. *RC* stands for release consistency. However, the only extra feature in this case is that the synchronization operations follow the *pc* (processor consistency) memory model instead of sequential consistency. Prove that for properly synchronized programs, *RCpc* leads to PC executions.

Ex. 20 — How do lazy and eager conflict detection mechanisms differ from each other? What is the effect of these schemes on the overall system performance?

Ex. 21 — Prove that the TL2 system algorithm is correct.

Ex. 22 — Can transactional memory systems suffer from livelocks? If yes, how do you prevent them.

* **Ex. 23** — Define opacity. How do we ensure opacity in STM systems? Does hardware transactional memory guarantee opacity?

Ex. 24 — When we want to commit a transaction in an STM, we lock all the locations that were written. Can this lead to deadlocks? If yes, how will you avoid deadlocks?

Design Problems

Ex. 25 — Understand the working of cache coherence protocols in the Tejas architectural Simulator.

Ex. 26 — Design a circuit to implement the directory protocol in Verilog or VHDL.

Ex. 27 — Understand the memory models of different programming languages such as C++ 17 and JAVA.

Ex. 28 — Download a popular STM library. Use it to write parallel programs.