Chapter 8

Hardware Support for Synchronization

Contents

As we have discussed in Chapter 6, hardware support for building a shared memory multiprocessor system that guarantees correct and efficient execution of parallel programs must address cache coherence, memory consistency, and support for synchronization primitives. The objective of this chapter is to discuss hardware support for implementing synchronization primitives. From the point of view of software, synchronization primitives that are widely used are locks, barriers, and point-to-point synchronizations such as signal-wait pairs. Recall for example that locks and barriers are heavily used in DOALL parallelism and in applications with linked data structures, whereas signal-wait synchronization is essential in pipelined (e.g., DOACROSS) parallelism.

There are many ways of supporting synchronization primitives in hardware. A common practice today is to implement the lowest level synchronization primitive in the form of atomic instructions in hardware, and implement all other synchronization primitives on top of that in software. We will discuss the trade-offs of various implementations of locks and barriers. We will show that achieving fast but scalable synchronization is not trivial. Often there are trade-offs in *scalability* (how synchronization latency and bandwidth scale with a larger number of threads) versus *uncontended latency* (the latency when threads do not simultaneously try to perform the synchronization). We

will also discuss several software barrier implementations. For very large systems, software barriers implemented on top of an atomic instruction and lock may not give sufficient scalability. For these systems, hardware barrier implementation is common, and we will discuss one example.

Finally, we will discuss transactional memory which is supported in recent multicore architecture. Transactional memory provides a higher abstraction level for coordinating parallel execution, in some cases removing the need to use lower level primitives such as locks. We will discuss one implementation of transactional memory in hardware.

8.1 Lock Implementations

In this section, we will discuss hardware support for lock implementations, and in the next section, we will discuss hardware support for barrier implementations.

8.1.1 Evaluating the Performance of Lock Implementations

Before discussing various lock implementations, first we will discuss what performance criteria need to be taken into account in a lock implementation. They are:

1. **Uncontended lock-acquisition latency**: the time it takes to acquire a lock when there is no contention between threads.

2. **Traffic**: the amount of traffic generated as a function of number of threads or processors that contend for the lock. Traffic can be subdivided into three: the traffic on *lock acquisition when a lock is free*, traffic on *lock acquisition when a lock is not free*, and traffic on *lock release*.

3. **Fairness**: the degree in which threads are allowed to acquire locks with respect to one another. One criteria related to fairness is whether in the lock implementation, it is possible for a thread to be *starved*, i.e., it is unable to acquire a lock for a long period of time even when the lock became free during that time.

4. **Storage**: the amount of storage needed as a function of number of threads. Some lock implementations require a constant storage space independent of the number of threads that share the lock, while others require a storage space that grows linearly along with the number of threads that share the lock.

8.1.2 The Need for Atomic Instructions

Recall the discussion in Chapter 6 in which we compare software versus hardware mechanisms for guaranteeing mutual exclusion. We have shown that a software mechanism (i.e., Peterson's algorithm) does not scale because the number of static instructions executed and the number of variables that need to be tested in order for a thread to check the condition for a lock acquisition increases along the number of threads. In contrast, if an *atomic instruction* that can perform a sequence of load, compare or other instructions, and a store, is available, a simple lock implementation that relies on testing only one variable is sufficient.

In current systems, most processors support an atomic instruction as (almost) the lowest level primitive on which other synchronization primitives can be built. An atomic instruction performs a

sequence of read, modify, and write in an indivisible manner as one indivisible operation. Consider the lock implementation in the following code (Code 8.1).

Code 8.1 An incorrect implementation of lock/unlock functions.

```
1lock: ld  R1, &lockvar   // R1 = lockvar
2      bnz R1, lock       // jump to lock if R1 != 0
3      st  &lockvar, #1   // lockvar = 1
4      ret               // return to caller
5
6unlock: st  &lockvar, #0 // lockvar = 0
7        ret              // return to caller
```

In order for the code to work correctly, the sequence of load, branch, and store instructions must be performed *atomically*. The term atomic implies two things. First, it implies that either *the whole sequence is executed entirely*, or none of it appears to have executed. Second, it also implies that *only one atomic instruction from any processor can be executed at any given time*. Multiple instruction sequences from several processors will have to be serialized. Certainly, software support cannot provide any atomicity of the execution of several instructions. Thus, most processors support atomic instructions, instructions that are executed atomically with respect to other processors. Different processors may support different types of atomic instructions. Below are some of the commonly used ones:

- `test-and-set Rx, M`: read the value stored in memory location M, test the value against a constant (e.g., 0), and if they match, write the value in register Rx to the memory location M.
- `fetch-and-op M`: read the value stored in memory location M, perform `op` to it (e.g., increment, decrement, addition, subtraction), and then store the new value to the memory location M. In some cases, an additional operand may be specified.
- `exchange Rx, M`: atomically exchange (or swap) the value in memory location M with the value in register Rx.
- `compare-and-swap Rx, Ry, M`: compare the value in memory location M with the value in register Rx. If they match, write the value in register Ry to M, and copy the value in Rx to Ry.

Out of the instructions listed above, the most versatile one is the compare and swap (CAS). Compared to the test-and-set, it is able to perform a comparison, but with an arbitrary value in a register operand instead of with a constant. Compared to an exchange, It can swap values in a register and a memory location, but with an attached condition. However, it cannot perform what fetch-and-op does except in some situations.

Two natural questions that readers may ask include: (1) how can atomicity be ensured for an atomic instruction? and (2) how can an atomic instruction be used to construct synchronization? We will discuss the answer to the first question first.

An atomic instruction essentially provides a guarantee to programmers that a sequence of operations represented by the instruction will be executed in its entirety. Using this guarantee, programmers can implement a variety of synchronization primitives they need (to be discussed later).

■ *Did you know?*

In the x86 instruction set, in addition to atomic instructions, regular integer instructions can also be made atomic by prefixing it with LOCK. When a memory location is accessed by an instruction that is LOCK-prefixed, assuming a bus-based multiprocessor, the LOCK# bus line is asserted to prevent other processors to read or modify the location while the prefixed instruction executes. For example, the following instructions can be LOCK-prefixed:

- *Bit manipulation instructions such as BT, BTS, BTR, and BTC.*
- *Arithmetic instructions such as ADD, ADC, SUB, and SBB.*
- *Logical instructions such as NOT, AND, OR, and XOR.*
- *Some unary instructions such as NEG, INC, and DEC.*

Atomic instructions are also provided. They start with "X", such as XADD, and XCHG. These atomic instructions do not require the "LOCK" prefix to make them atomic.

To illustrate the use of the prefix, let us say there is a variable `counter` *that we want to increment atomically. To do that, we can prefix the INC instruction with the LOCK prefix, as shown in the following example:*

```
1 void __fastcall atomic_increment (volatile int* ctr)
2 {
3     __asm {
4         lock inc dword ptr [ECX]
5         ret
6     }
7 }
```

The actual mechanism that guarantees atomicity of a sequence of operations itself must be provided correctly by the hardware.

Fortunately, the cache coherence protocol provides a foundation on which atomicity can be guaranteed. For example, when an atomic instruction is encountered, the coherence protocol knows that it must guarantee atomicity. It can do so by first obtaining an exclusive ownership of the memory location M (by invalidating all other cached copies of blocks that contain M). After the exclusive ownership is obtained, the protocol has ensured that only one processor has an access to the block (all other processors will suffer from a cache miss if they try to access the block). The atomic instruction can then be executed. For the duration of the atomic instruction, the block must not be allowed to be stolen by other processors. For example, if another processor requests to read or write to the block, the block is "stolen" (i.e., it is flushed and its state is downgraded). Exposing the block before the atomic instruction is completed breaks the atomicity of the instruction, similar to the non-atomic execution of a sequence of instructions in the naive lock implementation case (Code 8.1). Therefore, before the atomic instruction is completed, the block cannot be stolen. One way to prevent stealing of the block (on a bus-based multiprocessor) is to lock or reserve the bus until the instruction is complete. Since the bus is the serializing medium in the system, if it is locked, no other bus transactions can access the bus until it is released. A more general solution (which also applies to non-bus based multiprocessors) is not to prevent other requests from going to the bus, but for the coherence controller of the processor that executes the atomic instruction to defer responding to all requests to the block until the atomic instruction is completed, or negatively acknowledge the requests so that the requestors can retry the requests in the future. Overall, through

exclusive ownership of a block and preventing it from being stolen until the atomic instruction is complete, atomicity is ensured. A simpler solution to provide atomicity will be discussed later in Section 8.1.5.

8.1.3 Test and Set Lock

How is the atomic instruction used to implement a lock? It depends on the type of the atomic instruction. The following code illustrates a lock implementation using a test-and-set instruction (Code 8.2). The first instruction in the lock acquisition attempt is the atomic test-and-set instruction, which performs the following steps atomically: it reads from the memory location where `lockvar` is located (using an exclusive read such as BusRdX or BusUpgr), into register R1, compares R1 with zero. If it finds that R1 is 0, writes "1" to `lockvar` (indicating a successful lock acquisition). If R1 is not 0, it does not write "1" to `lockvar` (indicating a failed lock acquisition). The second instruction branches back to the label `lock` if R1 is not zero, so that the lock acquisition can be retried. If the value of R1 is zero, then we know that when the branch instruction is reached, the test-and-set instruction is successful due to its atomicity. Note that releasing the lock is simply performed by storing a zero to `lockvar` without using an atomic instruction. This works because there is only one thread in the critical section, so there is only one thread that can release the lock. Hence, no races can occur.

Code 8.2 An implementation of test-and-set lock.

```
1 lock: t&s R1, &lockvar // R1 = lockvar
2                        // if (R1==0) lockvar=1
3       bnz R1, lock     // jump to lock if R1 != 0
4       ret              // return to caller
5
6 unlock: st  &lockvar, #0 // lockvar = 0
7         ret              // return to caller
```

Let us see how the execution of two threads now allows only one thread to enter the critical section. Figure 8.1 illustrates the case. Let us assume that thread 0 executes the test-and-set instruction slightly earlier than P1. If the memory block where `lockvar` is stored initially has a value of 0, then after the execution of the test-and-set instruction, it holds a value of 1. Note that the atomicity of test-and-set serializes the execution of other test-and-set instructions. Hence, while test-and-set from thread 0 is ongoing, test-and-set from thread 1 is not allowed to proceed. When the test-and-set instruction of thread 1 executes, it sees the value of 1 in the memory location where `lockvar` is located. Hence, its lock acquisition fails, and thread 1 goes back to retry the test-and-set, on and on until it finally succeeds when the lock is released by thread 0.

Let us now evaluate the test-and-set lock implementation. The uncontended lock-acquisition latency is low since only one atomic instruction, plus a branch instruction, are needed to successfully obtain a lock. The traffic requirement, however, is very high. Each lock acquisition attempt causes invalidation of all cached copies, regardless of whether the acquisition is successful or not. For example, look at the example in Table 8.1, in which three threads running on three processors all trying to obtain a lock once. Initially, no one holds the lock. Then, P1 executes a test-and-set and successfully obtains the lock. The test-and-set instruction incurs a BusRdX bus transaction since the block is not already cached in P1's cache. Suppose that after P1 obtains the lock (while P1 is in

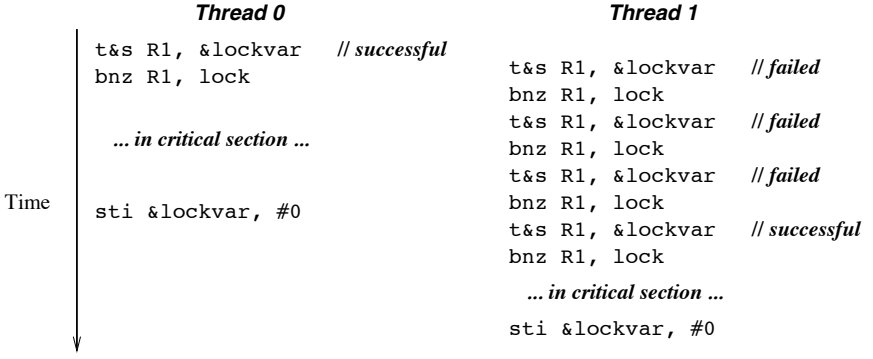

Figure 8.1: The execution of test-and-set atomic instructions from multiple processors.

the critical section), P2 and P3 attempt to acquire the lock. P2 executes a test-and-set, followed by an attempt by P3, and then by P2 for the second time. Each of these attempts involves a test-and-set instruction, so it incurs a BusRdX transaction to invalidate other cached copies and transitions the state of the block that contains `lockvar` to modified. Note that despite the fact that the lock is held by P1, P2 and P3 will keep incurring bus transactions to invalidate each other's copy in attempts to acquire the lock. This is one significant drawback of the test-and-set lock implementation. In fact, there may be many more failed attempts than shown in the table.

When later P1 releases the lock, it writes "0" to the lock variable, causing invalidation through the BusRdX transaction. Then, both P2 and P3 try to acquire the lock, but P2 succeeds and P3 fails. P3 keeps trying and failing to acquire the lock that is held by P2 for as long as P2 is in the critical section (the table shows only two failed attempts, but in reality they can occur many more times). After P2 releases the lock, P3 can acquire the lock successfully. Since no processor is contending for the lock now, when P3 releases the lock, it still has the cache block in modified state and does not incur a BusRdX bus transaction to store "0" to the lock address.

The table shows that a bus transaction is incurred at every attempt to acquire the lock, regardless of whether the lock is currently held or not. The amount of traffic for a single critical section that appears in the source code is quite high: $\mathcal{O}(p^2)$. This is because there are $\mathcal{O}(p)$ processors which will enter the critical section, which means that there are $\mathcal{O}(p)$ lock acquisitions and releases. After each lock release, there will be $\mathcal{O}(p)$ attempts to acquire the lock, with one processor succeeding in acquiring the lock, and the rest failing in acquiring the lock.

Obviously, the traffic incurred by the test-and-set lock can be excessively high. The traffic due to the lock acquisition attempts can slow down regular and coherence cache misses. Indeed, the critical section itself may be slowed down if traffic is saturated by failed lock acquisition attempts, delaying the lock to release, which further exacerbates the traffic situation.

One way to reduce the traffic requirement is to use a *back-off* strategy, in which after a failed lock acquisition attempt, a thread "waits" (or backs off) before performing another attempt. A delay can be inserted between subsequent retries. The delay in the back-off strategy needs to be carefully tuned: if it is too small, high amount of traffic remains; and if it is too large, the thread may miss an opportunity to acquire the lock when the lock becomes available. In practice, an *exponential back-off* strategy, in which the delay starts small but gradually increases exponentially, works quite well.

Table 8.1: Illustration of test-and-set lock implementation performance.

	Request	P1	P2	P3	Bus Request	Comments
0	Initially	–	–	–	–	lock is free
1	t&s1	M	–	–	BusRdX	P1 obtains lock
2	t&s2	I	M	–	BusRdX	P2 lock acq fails
3	t&s3	I	I	M	BusRdX	P3 lock acq fails
4	t&s2	I	M	I	BusRdX	P2 lock acq fails
5	unlock1	M	I	I	BusRdX	P1 releases lock
6	t&s2	I	M	I	BusRdX	P2 obtains lock
7	t&s3	I	I	M	BusRdX	P3 lock acq fails
8	t&s3	I	I	M	–	P3 lock acq fails
9	unlock2	I	M	I	BusRdX	P2 releases lock
10	t&s3	I	I	M	BusRdX	P3 obtains lock
11	unlock3	I	I	M	–	P3 releases lock

8.1.4 Test and Test and Set Lock

Another way to reduce the traffic requirement is to have a criteria that tests whether a lock acquisition attempt will likely lead to failure, and if that is the case, we defer the execution of the atomic instruction. Only when a lock acquisition attempt has a good chance of succeeding do we attempt the execution of the atomic instruction. Using this approach, the test-and-set lock can be improved and the improved version is referred to as the *test-and-test-and-set lock* (TTSL) implementation. The code for TTSL is shown in Code 8.3.

Code 8.3 An implementation of test-and-test-and-set lock (TTSL).

```
1lock: ld  R1, &lockvar // R1 = lockvar
2      bnz R1, lock     // jump to lock if R1 != 0
3      t&s R1, &lockvar // R1 = lockvar
4                       // if (R1 == 0) lockvar=1
5      bnz R1, lock     // jump to lock if R1 != 0
6      ret              // return to caller
7
8unlock: st  &lockvar, #0 // lockvar = 0
9        ret              // return to caller
```

The load instruction and the following branch instruction form a tight loop that keeps reading (but not writing) the address where `lockvar` is located until the value in the location turns 0. Therefore, while a lock is being held by a processor, other processors do not execute the test-and-set atomic instructions. This prevents repeated invalidations that are useless as they lead to failed lock acquisitions. Only when the lock is released, will the processors attempt to acquire the lock using the atomic instructions. The uncontended lock-acquisition latency is higher than the test-and-

set lock implementation due to the extra load and branch instructions. However, the traffic during the time a lock is held is significantly reduced. Only a small fraction of lock acquisition attempts generate traffic.

Table 8.2 illustrates the performance of the TTSL implementation. Suppose that each processor needs to get the lock exactly once. Initially, no one holds the lock. Then, P1 executes a load and finds out that the lock is available. So it attempts a test-and-set and successfully obtains the lock. The test-and-set instruction does not incur a BusRdX bus transaction since the block is exclusively cached. Suppose that after P1 obtains the lock (while P1 is in the critical section), P2 and P3 attempt to acquire the lock. Compared to the test-and-set lock implementation, in TTSL, P2 executes a load, followed by a load by P3. Both find that the value in `lockvar` location is one, indicating the lock is currently held by another processor. So they keep reading from the cache block, waiting until the value changes. While they are waiting, no bus transaction is generated since they are spinning using a load instruction rather than a test-and-set instruction. One of such loads is shown in line 5. In reality, it is likely that many loads are executed, rather than just one, so the advantage of TTSL over the test-and-set lock implementation increases. Note that an additional benefit of spinning using loads is that since bus bandwidth is conserved, the thread that is in the critical section does not experience the bandwidth contention that could slow it down.

Next, when P1 releases the lock, suppose that P2 is the first to suffer from a miss and discover that the lock has been freed; P2 then acquires the lock using the test-and-set instruction. This sequence produces two bus transactions: a bus read for loading the value, and a bus upgrade caused by the test-and-set instruction to write to the block. In contrast, in the test-and-set lock implementation, only one bus transaction is involved. Hence, the lock acquisition in TTSL incurs slightly more bandwidth usage than the test-and-set lock implementation, although overall, less bandwidth is used due to the use of loads for spinning.

Next, suppose that P3 spins on the lock variable (lines 9 and 10). When the lock is released by P2 (line 11), P3 attempts to read, and successfully acquires the lock, and succeeds. And finally, P3 releases the lock.

8.1.5 Load Linked and Store Conditional Lock

Although TTSL is an improvement over the test-and-set lock implementation, it still has a major drawback in its implementation. As mentioned before, one way to implement an atomic instruction requires a separate bus line that is asserted when a processor is executing an atomic instruction. Such an implementation is not general enough as it only works for bus-based multiprocessors. In addition, it works well only if lock frequency is low. If programmers use fine-grain locks, they use many lock variables simultaneously. There are many instances of lock acquisitions and releases, and many of them are unrelated as they are used, not for ensuring a critical section, but rather, for ensuring exclusive access to different parts of a data structure. If each of these lock acquisition asserts a single bus line, then unnecessary serialization occurs.

Another implementation, which reserves a cache block for the entire duration of the atomic instruction, is more general. It does not assume the existence of a special bus line so it can work with other interconnects as well. However, to prevent a cache block from being stolen by other processors, requests to the block must be deferred or negatively acknowledged. Such a mechanism can be costly to implement. Deferring requests requires an extra buffer to queue the requests, while negative acknowledgments waste bandwidth and incur delay when requests are retried in the future.

Table 8.2: Illustration of TTSL implementation performance.

	Request	P1	P2	P3	Bus Request	Comments
0	Initially	–	–	–	–	lock is free
1	ld1	E	–	–	BusRd	P1 reads from &lockvar
2	t&s1	M	–	–	–	P1 obtains lock
3	ld2	S	S	–	BusRd	P2 reads from &lockvar
4	ld3	S	S	S	BusRd	P3 reads from &lockvar
5	ld2	S	S	S	–	P2 reads from &lockvar
6	unlock1	M	I	I	BusUpgr	P1 releases lock
7	ld2	S	S	I	BusRd	P2 reads from &lockvar
8	t&s2	I	M	I	BusUpgr	P2 obtains lock
9	ld3	I	S	S	BusRd	P3 reads from &lockvar
10	ld3	I	S	S	–	P3 reads from &lockvar
11	unlock2	I	M	I	BusUpgr	P2 releases lock
12	ld3	I	S	S	BusRd	P3 reads from &lockvar
13	t&s3	I	I	M	BusUpgr	P3 obtains lock
14	unlock3	I	I	M	–	P3 releases lock

In order to avoid complexity associated with supporting an atomic instruction, an alternative is to provide an *illusion of atomicity* on a sequence of instructions, rather than true instruction atomicity. Note that the lock acquisition essentially consists of a load (`ld R1, &lockvar`), some instructions such as the conditional branch (`bnz R1, lock`), and a store (`st &lockvar, R1`). Atomicity implies that either none or all of the instructions appear to have executed. From the point of view of other processors, only the store instruction is visible, as it changes a value that may be seen by them. Other instructions (the load and the branch) have their effect on the registers of the local processor but their effect is not visible to other processors. Hence, from the point of view of other processors, it does not matter if the load and the branch have executed or not. From the point of view of the local processor, these instructions can be canceled easily by ignoring their register results and reexecuting them later. Therefore, the instruction that is critical to the illusion of atomicity is the store instruction. If it is executed, its effect is visible to other processors. If it is not executed, its effect is not visible to other processors. Therefore, in order to give the illusion of atomicity to the sequence of instructions, we need to ensure that the store fails (or gets canceled) if between the load and the store, something has happened that potentially violates the illusion of atomicity. For example, if there is a context switch or interrupt that occurs between the load and the store, then the store must be canceled (servicing the interrupt may result in the value of the block to change without the knowledge of the processor, which breaks atomicity). In addition, with respect to other processors, if the cache block that is loaded has been invalidated before the store executes, then the value in the cache block may have changed, resulting in a violation to atomicity. In this

case, the store must be canceled. If, however, the block remains valid in the cache by the time the store executes, then the entire load-branch-store sequence can appear to have executed atomically.

The illusion of atomicity requires that the store instruction is executed conditionally upon detected events that may break the illusion of atomicity. Such a store is well known as a *store conditional* (SC). The load that requires a block address to be monitored from being *stolen* (i.e., invalidated) is known as a *load linked* or *load locked* (LL). The LL/SC pair turns out to be a very powerful mechanism on which many different atomic operations can be built. The pair ensures the illusion of atomicity without requiring an exclusive ownership of a cache block.

An LL is a special load instruction that not only reads a block into a register, but also records the address of the block in a special processor register which we will refer to as a *linked register*. An SC is a special store instruction that succeeds only when the address involved matches the address stored in the linked register. To achieve the illusion of atomicity, the SC should fail when another processor has raced past it, successfully performed an SC, and stolen the cache block. To ensure the failure of SC, the linked register is cleared when an invalidation to the address stored in the linked register occurs. In addition, when a context switch occurs, the linked register is also cleared. When an SC fails (due to mismatch of the SC address and the address in the linked register), the store is canceled without going to the caches. Hence, to the memory system, it is as if the SC is never executed. All other instructions in the atomic sequence (LL included) can simply be repeated as if they have also failed.

Code 8.4 shows the code that implements a lock acquisition and release using an LL/SC pair, which is identical to Code 6.3 except that the load is replaced with the LL, and the store is replaced with an SC.

Code 8.4 An implementation of LL/SC lock. The code assumes that a failed SC returns a 0.

```
1 lock: LL   R1, &lockvar  // R1 = lockvar;
2                          // LINKREG = &lockvar
3       bnz  R1, lock      // jump to lock if R1 != 0
4       add  R1, R1, #1    // R1 = 1
5       SC   &lockvar, R1  // lockvar = R1;
6       beqz R1, lock      // jump to lock if SC fails
7       ret                // return to caller
8
9 unlock: st  &lockvar, #0 // MEM[&lockvar] = 0
10        ret              // return to caller
```

The LL instruction and the following branch instruction forms a tight loop that keeps reading (but not writing) the location where `lockvar` is until the value in the location turns 0. Thus it behaves similarly to a TTSL implementation which spins using loads. Therefore, while a lock is being held by a processor, the SC is not executed. This prevents repeated, useless, invalidations that correspond to failed lock acquisitions. Only when the lock is released, will the processors attempt to acquire the lock using the SC instruction. It is possible that multiple processors attempt to perform the SC simultaneously. On a bus-based multiprocessor, one of the SCs will be granted a bus access and performs the store successfully. Other processors snoop the store on the bus and clear out their linked registers, causing their own SC to fail. In contrast to a test-and-set atomic instruction, when an SC fails, it does not generate a bus transaction (a test-and-set instruction always generates a BusRdX or BusUpgr).

■ *Did you know?*

LL/SC instruction pairs are supported in various instruction sets such as Alpha (`ldl_l/stl_c` *and* `ldq_l/stq_c`*), PowerPC (*`lwarx/stwcx`*), MIPS (*`LL/SC`*), and ARM (*`ldrex/strex`*). The conditions that make SC fails can vary across implementations although sometimes the failure is not strictly necessary given the conditions. For example, in some implementations, SC fails when another LL is encountered, or even when ordinary loads or stores are encountered.*

Table 8.3 illustrates the performance of the LL/SC lock implementation. Suppose that each processor needs to get the lock exactly once. Initially, no one holds the lock. The sequence of bus transactions generated is identical to ones in the TTSL implementation, except that the LL replaces the regular load, and SC replaces the test-and-set instruction.

Table 8.3: Illustration of LL/SC lock implementation performance.

	Request	**P1**	**P2**	**P3**	**Bus Request**	**Comments**
0	Initially	–	–	–	–	lock is free
1	ll1	E	–	–	BusRd	P1 reads from &lockvar
2	sc1	M	–	–	–	P1 obtains lock
3	ll2	S	S	–	BusRd	P2 reads from &lockvar
4	ll3	S	S	S	BusRd	P3 reads from &lockvar
5	ll2	S	S	S	–	P2 reads from &lockvar
6	unlock1	M	I	I	BusUpgr	P1 releases lock
7	ll2	S	S	I	BusRd	P2 reads from &lockvar
8	sc2	I	M	I	BusUpgr	P2 obtains lock
9	ll3	I	S	S	BusRd	P3 reads from &lockvar
10	ll3	I	S	S	–	P3 reads from &lockvar
11	unlock2	I	M	I	BusUpgr	P2 releases lock
12	ll3	I	S	S	BusRd	P3 reads from &lockvar
13	sc3	I	I	M	BusUpgr	P3 obtains lock
14	unlock3	I	I	M	–	P3 releases lock

Therefore, performance-wise, the LL/SC lock implementation performs similarly to a TTSL implementation. A minor difference is when multiple processors simultaneously perform SCs. In this case, only one bus transaction occurs in LL/SC (due to the successful SC), whereas in the TTSL, there will be multiple bus transactions corresponding test-and-set instruction execution. However, this is a quite rare event since there are only a few instructions between a load and a store in an atomic sequence, so the probability of multiple processors executing the same sequence at the same time is small.

However, the advantages of LL/SC over using atomic instructions are numerous. First, LL/SC implementation is relatively simpler (extra linked registers). Secondly, it can be used to implement many atomic instructions, such as test-and-set, compare-and-swap, etc. Hence, it can be thought of as a lower level primitive than atomic instructions.

However, the LL/SC lock implementation still has a significant scalability problem. Each lock acquisition or lock release triggers invalidation of all sharers that spin on the lock variable because the value of the lock variable has changed. They in turn suffer from cache misses to reload the block containing the lock variable. Hence, if there are $\mathcal{O}(p)$ lock acquisitions and releases, and each acquisition or release triggers $\mathcal{O}(p)$ subsequent cache misses, then the total traffic of an LL/SC lock implementation scales quadratically to the number of threads in the system, or $\mathcal{O}(p^2)$. This is in addition to the issue of lock fairness in which there is no guarantee that the thread that attempts to acquire the lock the earliest can actually acquire the lock earlier than others. The following two lock implementations will address each of these problems.

8.1.6 Ticket Lock

The *ticket lock* is a lock implementation that attempts to provide fairness in lock acquisition. The notion of fairness deals with whether the order in which threads first attempt to acquire a lock corresponds to the order in which threads acquire the lock successfully. This notion of fairness automatically guarantees that a thread is not starved out from failing to acquire a lock for a long period of time because other threads always beat to it.

To achieve such fairness, the ticket lock implementation uses a concept of a queue. Each thread that attempts to acquire a lock is given a ticket number in the queue, and the order in which lock acquisition is granted is based on the ticket numbers, with a thread holding the lowest ticket number is given the lock next.

To implement that, two variables are introduced. The first one is `now_serving` and the other is `next_ticket`. The role of `next_ticket` is to reflect the next available ticket number (or the order position) that a new thread should get. The role of `now_serving` is to reflect the position of the current holder of the lock. Necessarily, $next\_ticket - now\_serving - 1$ is the number of threads that are currently waiting for the lock, i.e., they have attempted to acquire the lock but have not obtained it. To obtain a lock, a thread reads the `next_ticket` into a private variable (say, `my_ticket`) and atomically increments it, so that future threads will not share the same ticket number. Then, it waits until `now_serving` is equal to the value of `my_ticket`. A thread that is currently holding the lock releases the lock by incrementing the `now_serving` variable so that the next thread in line will find that the new value of `now_serving` is equal to its `my_ticket`, and the thread can successfully acquire the lock. The complete code for lock acquisition and release are shown in Code 8.5.

Note that the code shows the implementation in a high-level language, assuming that there is only one lock (so a lock's name is not shown) and fetch the atomic primitive `fetch_and_inc()` is supported. As we have seen in the discussion of LL/SC lock implementation, such primitive can be implemented easily using a pair of LL and SC. The only thing that needs to be inserted between the LL and SC is an instruction that increments the value read from memory by LL. We also assume that `fetch_and_inc()` is implemented as a function that returns the old value of the argument prior to incrementing it. Finally, the values of `now_serving` and `next_ticket` are both initially initialized to 0 before use.

Code 8.5 Ticket lock implementation.

```
1 ticketLock_init(int *next_ticket, int *now_serving)
2 {
3   *now_serving = *next_ticket = 0;
4 }
5
6 ticketLock_acquire(int *next_ticket, int *now_serving)
7 {
8   my_ticket = fetch_and_inc(next_ticket);
9   while (*now_serving != my_ticket) {};
10 }
11
12 ticketLock_release(int *now_serving)
13 {
14   now_serving++;
15 }
```

To illustrate how the ticket lock works, consider three processors contending for the lock and each of them will acquire and release the lock exactly once. Table 8.4 shows the values of the important variables at each step: `now_serving`, `next_ticket`, P1's `my_ticket`, P2's `my_ticket`, and P3's `my_ticket`. First, P1 tries to acquire lock, it executes `fetch_and_inc(next_ticket)` atomically, resulting in the value of `next_ticket` to increase to 1. It then compares `my_ticket` and `now_serving`, and since their values are equal, it successfully acquires the lock and enters the critical section. P2, contending for the lock, also attempts to acquire the lock by executing `fetch_and_inc (next_ticket)`, resulting in the `next_ticket` to increase to 2. It then compares `my_ticket`, which has a value of 1, and `now_serving`, which has a value of 0 (since the lock is still held by the thread having the ticket number 0). Since their values are not equal, P2 fails the lock acquisition, stays in the loop to keep on testing the values of `now_serving`. P3, also contending for the lock, attempts to acquire the lock by executing `fetch_and_inc(next_ticket)`, resulting in the `next_ticket` to increase to 3. It then compares `my_ticket`, which has a value of 2, and `now_serving`, which still has a value of 0 (since the lock is still held by P1). Since their values are not equal, P3 fails the lock acquisition, and like P2, it stays in the loop to keep on testing their values. When P1 releases the lock, it executes the `now_serving++` statement, incrementing the `now_serving` to a value of 1. Now P2's `my_ticket` and `now_serving` match, P2 gets out of its loop and enters the critical section. Similarly, when P2 releases the lock, it executes the `now_serving++` statement, incrementing the `now_serving` to a value of 2. Now P3's `my_ticket` and `now_serving` match, P3 gets out of its loop and enters the critical section.

The performance of ticket lock depends on how the `fetch_and_inc` atomic operation is implemented. If it is implemented with an atomic instruction, then its scalability is similar to that of the atomic instruction. If the `fetch_and_inc` is implemented using an LL/SC pair, then its scalability is similar to that of LL/SC lock implementation. In particular, the `fetch_and_inc` on `next_ticket` will cause a processor to invalidate all copies of the block containing `next_ticket`, and each processor subsequently suffers from a cache miss to reload the block. Similarly, since all processors are monitoring and spinning on the `now_serving`, when a processor releases the lock, all copies of the block containing `now_serving` are invalidated, and each processor subsequently suffers from a cache miss to reload the block. Since there are $\mathcal{O}(p)$ number of acquisition and

Table 8.4: Illustration of the ticket lock mechanism.

Steps	next_ ticket	now_ serving	**P1**	**P2**	**P3**	**Comments**
			my_ticket			
Initially	0	0	0	0	0	all initialized to 0
P1: f&i(next_ticket)	1	0	0	0	0	P1 tries to acq lock
P2: f&i(next_ticket)	2	0	0	1	0	P2 tries to acq lock
P3: f&i(next_ticket)	3	0	0	1	2	P3 tries to acq lock
P1: now_serving++	3	1	0	1	2	P1 rels, P2 acqs lock
P2: now_serving++	3	2	0	1	2	P2 rels, P3 acqs lock
P3: now_serving++	3	3	0	1	2	P3 rels lock

releases, and each acquire and release causes $\mathcal{O}(p)$ invalidations and subsequent cache misses, the total traffic scales on the order of $\mathcal{O}(p^2)$.

The uncontended latency of a ticket lock implementation is slightly higher than that of LL/SC because it has extra instructions that read and test the value of `now_serving`. However, the ticket lock provides fairness, which is not provided by the LL/SC lock implementation.

8.1.7 Array-Based Queuing Lock

At this point, readers may wonder if there is a lock implementation that has the fairness of the ticket lock and at the same time better scalability compared to all the previous lock implementations discussed so far. One answer to that lies in a lock implementation called the *array-based queuing lock* (ABQL). ABQL is an improvement over the ticket lock. It starts out from the observation that since threads already queue up waiting for the lock acquisition in the ticket lock, they can wait and spin on unique memory locations rather than on a single location represented by `now_serving`. The analogy for the new mechanism is for the current lock holder to pass the baton to the next thread in the queue, and for that thread to pass the baton to the next next thread, and so on. Each time the baton is passed, only the next thread needs to be notified.

To implement ABQL, we need to ensure that threads spin on unique memory locations. To achieve that, we can change the `now_serving` to an array, which we will rename to `can_serve` to reflect its role better. As with the ticket lock implementation, each thread that wants to acquire a lock obtains a ticket number, say x. It then waits in a loop until the x^{th} element in the `can_serve` array has a value of 1, which will be set by the thread ahead of it in the queue. When a thread with a ticket number y releases a lock, it passes the baton to the next thread by setting the $(y+1)^{th}$ element in the `can_serve` array to 1. The complete code for lock acquisition and release are shown in Code 8.6.

Note that the code shows the implementation in a high-level language, assuming that there is only one lock and the atomic primitive `fetch_and_inc()` is supported. Initially, the values of `next_ticket` are initialized to 0. Elements of the array `can_serve` are initialized to 0 as well, except for the first element which is initialized to 1, in order to allow the first lock acquisition to be successful.

Code 8.6 Array-based queuing lock implementation.

```
1 ABQL_init(int *next_ticket, int *can_serve)
2 {
3   *next_ticket = 0;
4   for (i=1; i<MAXSIZE; i++)
5     can_serve[i] = 0;
6   can_serve[0] = 1;
7 }
8
9 ABQL_acquire(int *next_ticket, int *can_serve)
10 {
11   *my_ticket = fetch_and_inc(next_ticket);
12   while (can_serve[*my_ticket] != 1) {};
13 }
14
15 ABQL_release(int *can_serve)
16 {
17   can_serve[*my_ticket + 1] = 1;
18   can_serve[*my_ticket] = 0;   // prepare for next time
19 }
```

To illustrate how ABQL works, consider three processors contending for the lock and each of them will acquire and release the lock exactly once. Table 8.5 shows the values of the important variables at each step: `can_serve` (assuming it has four elements), `next_ticket`, P1's `my_ticket`, P2's `my_ticket`, and P3's `my_ticket`.

First, P1 tries to acquire lock, it executes `fetch_and_inc(next_ticket)` atomically, resulting in the value of `next_ticket` to increase to 1, and it obtains a value of 0 for its `my_ticket`. Then it checks whether `can_serve[my_ticket]` (`can_serve[0]`) is 1. Since the value is 1, it successfully acquires the lock and enters the critical section. P2, contending for the lock, also attempts to acquire the lock by executing `fetch_and_inc(next_ticket)`, resulting in the `next_ticket` to increase to 2, and it obtains the value of 1 for its `my_ticket`. It then checks whether `can_serve[1]==1`. But at this point `can_serve[1]` has a value of 0 since P1 has not released its lock. So P2 fails the lock acquisition and stays in the loop. P3, also contending for the lock, also attempts to acquire the lock by executing `fetch_and_inc(next_ticket)`, resulting in the `next_ticket` to increase to 3, and it obtains the value of 2 for its `my_ticket`. It then checks whether `can_serve[2]==1`. But at this point `can_serve[2]` has a value of 0 since P1 has not released the lock (P1 has not even acquired the lock). So P3 fails the lock acquisition, stays in the loop. When P1 releases the lock, it resets `can_serve[0]` to 0 and sets `can_serve[1]` to 1, allowing P2 to get out of its loop and enter the critical section. When P2 releases the lock, it resets `can_serve[1]` to 0 and sets `can_serve[2]` to 1, allowing P3 to get out of its loop and enter the critical section. Finally, P3 releases the lock by setting `can_serve[3]` to 1, and resets `can_serve[2]` to 0.

Consider the way a thread releases a lock. It writes to one element of the array `can_serve[my_ticket+1]`. Assume that the array is padded such that different array elements reside in different cache blocks. Since exactly one thread spins in a loop reading from the element, the element is only stored in one cache. The write only invalidates one cache block, and only one subsequent cache miss occurs. Hence, on each release, only $\mathcal{O}(1)$ traffic is generated. That means

Table 8.5: Illustration of ABQL mechanism.

Steps	next_ ticket	can_ serve	**P1**	**P2**	**P3**	**Comments**
			my_ticket			
Initially	0	[1, 0, 0, 0]	0	0	0	all initialized to 0
P1: f&i(next_ticket)	1	[1, 0, 0, 0]	0	0	0	P1 tries to acq lock
P2: f&i(next_ticket)	2	[1, 0, 0, 0]	0	1	0	P2 tries to acq lock
P3: f&i(next_ticket)	3	[1, 0, 0, 0]	0	1	2	P3 tries to acq lock
P1: can_serve[1]=1; can_serve[0]=0	3	[0, 1, 0, 0]	0	1	2	P1 rels, P2 acqs lock
P2: can_serve[2]=1; can_serve[1]=0	3	[0, 0, 1, 0]	0	1	2	P2 rels, P3 acqs lock
P3: can_serve[3]=1; can_serve[2]=0	3	[0, 0, 0, 1]	0	1	2	P3 rels lock

that since there are $\mathcal{O}(p)$ number of acquisitions and releases, the total lock release traffic scales on the order of $\mathcal{O}(p)$, a significant improvement over the ticket lock implementation. However, note that the `fetch_and_inc`'s scalability depends on its underlying implementation. If it is implemented with an LL/SC, its scalability may partially restrict the overall scalability of the ABQL implementation.

8.1.8 Qualitative Comparison of Lock Implementations

Table 8.6 compares the various lock implementations based on several criteria: uncontended latency, the amount of traffic following a single lock release, the amount of traffic waiting while the lock is held by a processor, storage overheads, and whether fairness guarantee is provided.

Table 8.6: Comparison of various lock implementations.

Criteria	**test&set**	**TTSL**	**LL/SC**	**Ticket**	**ABQL**
Uncontended latency	Lowest	Lower	Lower	Higher	Higher
1 Release max traffic	$\mathcal{O}(p)$	$\mathcal{O}(p)$	$\mathcal{O}(p)$	$\mathcal{O}(p)$	$\mathcal{O}(1)$
Wait traffic	High	–	–	–	–
Storage	$\mathcal{O}(1)$	$\mathcal{O}(1)$	$\mathcal{O}(1)$	$\mathcal{O}(1)$	$\mathcal{O}(p)$
Fairness guarantee?	No	No	No	Yes	Yes

The uncontended latency is the lowest on simpler lock implementations such as the test_and_set, TTSL, and LL/SC. Ticket lock and ABQL implementations execute more instructions so their uncontended lock acquisition latency is higher.

In terms of the traffic following a single lock release, assuming all other threads are waiting to acquire the lock next, test_and_set, TTSL, LL/SC, and ticket lock have the highest amount of

maximum traffic. This is because all threads spin on the same variable so they may all cache the same block, and each lock release invalidates all other processors and forces them to suffer misses to reload the block. On the other hand, in ABQL, a single release only invalidates one other cache causing only one subsequent cache miss.

In terms of the traffic generated while a lock is held by a thread, test_and_set performs very poorly because all threads keep on attempting to acquire the lock using an atomic instruction and even a failed attempt still generates an invalidation of all sharers and subsequent acquisition attempts by the sharers. On the other hand, TTSL, LL/SC, ticket lock, and ABQL, spin wait using a load instruction. Hence, once a thread discovers that the lock is held, it does not attempt to execute an atomic instruction.

In terms of storage requirements, all of the lock implementations use one or two shared variables so the storage requirement is constant across number of processors. Only ABQL has a storage requirement that scales with the number of processors due to keeping the array `can_serve`.

In terms of guarantee of fairness, only the ticket lock and ABQL provide it, due to the use of a queue. In other implementations, it is possible (though in reality may not be likely) that a processor has a better chance at acquiring the lock more than others following a release. For example, that may occur when a thread that releases a lock (and invalidating other caches) quickly acquires the lock to reenter the critical section again. In the mean time, other processors have not even had a chance to reattempt a lock acquisition because they are still reloading the invalidated block. In this case, it is possible that a thread keeps on acquiring and releasing a lock at the expense of other threads' ability to acquire the lock. Other causes may also be possible. For example, if the bus arbitration logic favors some requesters than others, some processors may be granted the bus faster than others. In a distributed shared memory system, one processor may reload an invalidated block faster than others, for example due to reloading the block from its local memory whereas others must reload it from a remote memory. Following a release, the processor that reloads the block from its local memory may race past others to attempt a lock acquisition.

Note, however, fairness guarantee creates a risk in performance. If a thread that is already in the queue waiting to get a lock is context switched out, then even when the lock becomes available, the thread will not attempt to acquire the lock. Other threads with larger ticket numbers cannot acquire the lock either because they must wait until the thread that is switched out has acquired and released the lock. Thus, the performance of all threads are degraded by the context switch of one of the threads. Therefore, care must be taken to ensure context switches do not occur when using the ticket lock or ABQL implementations.

From the point of view of software, it is not immediately clear which lock implementation is the best. For software showing a high degree of lock contention, ABQL offers the highest scalability. However, a high degree of lock contention is often a symptom of a deeper scalability problem, such as the use of lock granularity that is too coarse grain. In such a case, using ABQL improves scalability but may not make the program scalable. Better scalability can be obtained using a finer lock granularity. For example, in the linked list parallelization discussed in Chapter 4, using the fine grain lock approach, in which each node is augmented with its own lock, will likely make the contention level of any particular lock very low. In that case, ABQL is not only unnecessary, it is not even preferred due to a high uncontended latency and the fact that the storage overhead of one array for each node will be too high.

8.2 Barrier Implementations

Barriers are very simple and widely used synchronization primitives in parallel programs. We have discussed in Chapter 3 that in loop-level parallelization, barriers are often used at the end of a parallel loop to ensure all threads have computed their parts of the computation before the computation moves on to the next step. In the OpenMP `parallel for` directive, a barrier is automatically inserted at the end of a parallel loop, and programmers must explicitly remove it if they believe that the lack of barrier does not impact the correctness of their computation.

In this section, we will look at how barriers can be implemented, and compare the characteristics of the implementations in various terms. Barriers can be implemented in software or directly in hardware. Software barrier implementation is flexible but is often inefficient, whereas hardware barrier implementation restricts the flexibility and portability of the software but is often a lot more efficient. The simplest software barrier implementation is an implementation called the *sense-reversal global barrier*. The key mechanism used is for threads to enter the barrier and spin on a location that will be set by the last thread that arrives at the barrier, releasing all threads that are waiting there. Obviously, spinning on a single location often involves a lot of invalidations when the location is written, restricting scalability. One of the more complex but more scalable barrier implementations is *combining tree barrier* in which the threads that participate in a barrier are organized like a tree, and a thread at a particular tree level spins on a common location only with its siblings. This restricts the number of invalidations as threads spin on different locations. Finally, we will look at a hardware barrier implementation support that is implemented in some very large machine with thousands of processors.

The criteria for evaluating barrier performance include:

1. **Latency**: the time spent from entering the barrier to exiting the barrier. The latency in the barrier should be as small as possible.

2. **Traffic**: the amount of bytes communicated between processors as a function of the number of processors.

In contrast to lock implementation, fairness and storage overheads are not important issues because barriers are a global construct involving many threads.

8.2.1 Sense-Reversing Centralized Barrier

A software barrier implementation only assumes that lock acquisition and release primitives to be provided by the system (through software, hardware, or a combination of hardware and software). For example, as long as one of test-and-set, TTSL, LL/SC, ticket, or ABQL lock implementation is available, a barrier implementation can be constructed as well.

We note that from the point of view of programmers, a barrier must be simple, that is, a barrier should not require any arguments passed to it, either variable names or number of processors or threads. In OpenMP standard, for example, a barrier can be invoked simply as `#pragma omp barrier`. In the actual implementation, arguments may be used, as long as they are not exposed to the programmers.

The basic barrier implementation is shown in Code 8.7. The implementation makes use of several variables. `barCounter` keeps track of how many threads have so far arrived at the barrier.

barLock is a lock variable to protect a modification to shared variables in a critical section. canGo is a flag variable that threads spin on to know whether they can go past the barrier or not yet. Hence, canGo is set by the last thread to release all threads from the barrier.

Code 8.7 Simple (but incorrect) barrier implementation.

```
1// declaration of shared variables used in a barrier
2// and their initial values
3int numArrived = 0;
4lock_type barLock = 0;
5int canGo = 0;
6
7// barrier implementation
8void barrier () {
9  lock(&barLock);
10    if (numArrived == 0) { // first thread sets flag
11      canGo = 0;
12    }
13    numArrived++;
14    myCount = numArrived;
15  unlock(&barLock);
16
17  if (myCount < NUM_THREADS) {
18    while (canGo == 0) {};   // wait for last thread
19  }
20  else {  // last thread to arrive
21    numArrived = 0;  // reset for next barrier
22    canGo = 1;         // release all threads
23  }
24}
```

For example, suppose that three threads P1, P2, and P3 arrive at the barrier in that order. P1 arrives at the barrier first and enters the critical section. Then it initializes the variable canGo to 0 so that itself and other threads will wait until the last thread arrives and sets it to 1. It then increments numArrived to 1, assigns its value to myCount, and exits the critical section. Next, it enters a loop to wait until canGo has a value of 1. The second thread P2 enters the barrier, increments numArrived to 2, discovers that it is not the last thread (myCount has a value of 2, smaller than NUM_THREADS) and also enters the loop to wait until canGo has a value of 1. Finally, the last thread P3 enters the barrier, and increments numArrived to 3. It discovers that myCount is equal to NUM_THREADS so it is the last thread to arrive at the barrier. It then resets the value of numArrived to 0 so that it can be used in the next barrier. Then, it releases all waiting threads by setting canGo to 1. This allows all waiting threads to get out of the loop and resume computation past the barrier.

Unfortunately, the code described above does not work correctly across more than one barrier. For example, when the last thread P3 sets the value of canGo to 1, it invalidates all cached copies of the block where canGo is located. Following the invalidations, the block resides in P3's cache, while P1 and P2 try to reload the block. Suppose that before that can happen, P3 enters the next barrier, now as the first thread that arrives at that second barrier. As the first thread, it resets the variable canGo to 0. It can do so quickly since it has the block in its cache in a modified state. When P1 and P2 reload the block, they discover that the value of canGo is 0 as affected by the

second barrier rather than 1 as released from the first barrier. Therefore, P1 and P2 stay in the loop of the first barrier and never get released, while P3 waits in the loop of the second barrier, never to be released either.

One possible solution to the "deadlock" situation discussed earlier is to have a two-step release, in which before the first thread that enters a barrier initializes the `canGo` variable, it waits until all threads have been released from the previous barrier. Implementing such a solution requires another flag, another counter, and extra code, which can be quite costly. Fortunately, there is a simpler solution based on an observation that the error arises when the first thread entering the next barrier resets the value of `canGo`. If we avoid this reset, then the thread that enters the next barrier will not prevent other threads from getting out of the first barrier. To avoid resetting the value of `canGo`, in the second barrier, threads can instead wait until the value of `canGo` changes back to 0, which is accomplished by the last thread that enter the second barrier. With this approach, the value of `canGo` that releases threads alternates from 1 in the first barrier, 0 in the second barrier, 1 in the third barrier, 0 in the fourth barrier, and so on. Because the value is toggling between barriers, this solution is referred to as the *sense-reversing centralized barrier*. The code for the barrier is shown in the following (Code 8.8).

Code 8.8 Sense-reversing barrier implementation.

```
1 // declaration of shared variables used in a barrier
2 int numArrived = 0;
3 lock_type barLock = 0;
4 int canGo = 0;
5
6 // thread-private variables
7 int valueToWait = 0;
8
9 // barrier implementation
10 void barrier () {
11   valueToWait = 1 - valueToWait; // toggle it
12   lock(&barLock);
13     numArrived++;
14     myCount = numArrived;
15   unlock(&barLock);
16
17   if (myCount < NUM_THREADS) {  // wait for last thread
18     while (canGo != valueToWait) {};
19   }
20   else {  // last thread to arrive
21     numArrived = 0;  // reset for next barrier
22     canGo = valueToWait;   // release all threads
23   }
24 }
```

The code shows that each thread, when entering a barrier, first toggles the value that it will wait from 0 to 1 or from 1 to 0. Then, it increments the counter and waits until the value of `canGo` is changed by the last thread. The last thread is the one that toggles the value of `canGo`.

The centralized (or global) barrier implementation uses a critical section inside the barrier routine so as the number of threads grow, the time in the barrier increases linearly. Actually, it may increase super-linearly depending on the underlying lock implementation. In addition, the traffic is

high since each thread increments the variable `numArrived`, invalidating all sharers of the block, which then have to reload the block through cache misses. Since there are $\mathcal{O}(p)$ such increments, and each increment can cause $\mathcal{O}(p)$ cache misses, the total traffic for the barrier implementation scales quadratically with the number of processors, or $\mathcal{O}(p^2)$. Hence, unfortunately, the centralized barrier is not very scalable.

8.2.2 Combining Tree Barrier

There have been many attempts to improve the scalability of software barriers. In these scalable barriers, the situation in which all threads sharing and spinning on a single location (e.g., `barCount` or `canGo`) is avoided. The way they avoid spinning on a single location is by organizing the barrier in a hierarchical manner, where groups of threads synchronize within each group, one thread from each group is chosen to advance to the next round and forms a new group with other selected threads, and so on until the last group has finished synchronizing in the barrier. There are several scalable barrier algorithms, including combining tree barrier, tournament barrier, dissemination barrier, etc. that have been proposed in the past. We will discuss one such barrier: combining tree barrier.

The combining tree barrier divides the nodes (or threads) into subgroups with k members. Each group of threads synchronize a simple shared counter, for example by requiring each thread to atomically increment the counter and then wait until all threads in the group reach the barrier (counter value reaches k). Afterward, the first threads from each group form a new group of k members (representing a parent node) and synchronize again. This is repeated until there is one group left representing the root node of the tree. Finally, the root releases all threads by setting a flag that all threads monitor.

A combining tree barrier needs the storage space that is equivalent to the size of the tree, that is, $\mathcal{O}(p)$. The amount of traffic scales with the number of nodes, i.e., $\mathcal{O}(p)$, which is favorable compared to a centralized barrier's $\mathcal{O}(p^2)$. However, the latency is now higher, since to know that all threads have reached the barrier, we need to traverse up the tree, participating in $\mathcal{O}(log\ p)$ barriers at various levels of the tree, versus the centralized barrier's $\mathcal{O}(1)$.

8.2.3 Hardware Barrier Implementation

Hardware barrier implementation is attractive for its low latency as well as its scalability. With software barrier, we have to execute many instructions to implement a barrier primitive and rely on cache coherence mechanism to propagate changes made in the primitive. The cache coherence alone is at the moment unable to scale to thousands or tens of thousands of processors. In contrast, hardware implementation relies on signals traveling on dedicated wires. For a large system, the wires form a dedicated barrier network. The requirement of dedicated wires and network make hardware barrier implementations seem unnecessary for a small multiprocessor system, especially ones that are general purpose. However, for a large multiprocessor system, hardware barrier offers the only way to achieve truly scalable barrier implementations.

The simplest hardware barrier implementation in a bus-based multiprocessor is a special bus line that implements an AND logic. Each processor that arrives at the barrier asserts the barrier line. Since the barrier implements an AND logic, the line is only high when all processors have reached the barrier and asserted the line. Each processor also monitors the barrier line to detect its signal value. When processors sense that the barrier line is high, they conclude that all processors have

reached the barrier. At this point, they can exit the barrier and continue execution.

Conceptually, a hardware barrier network is simple. It needs to collect each signal from a processor that has reached the barrier, until signals from all processors have been collected. Then it needs to broadcast a barrier completion signal to all processors. All these need to be performed in as short time as possible. Signals cannot be collected by one node because it requires a very large connectivity. A limited connectivity requires the wires used for barrier signaling to form a network. An example of a scalable network with limited node connectivity is a tree. A k-ary tree allows a node to collect signals from k children nodes (unless the node is a leaf node), before passing a barrier signal to to its parent node. A node that is designated as the root eventually receives signals from all its children and detects the completion of the barrier. The root can then send a barrier completion signal down to its k children nodes, which then propagate the signal down to their children, and so on, all the way until the signal reaches the leaf nodes. The number of steps needed to collect signals from N processors will then be $log_k(N)$. The number of steps determine the latency to collect and broadcast barrier signals. With a log function, the latency increases more slowly than the size of the system, making the network a scalable barrier implementation.

■ *Did you know?*

IBM BlueGene supercomputer, designed to scale up to as many as 65 thousand processors, is an example system that requires a scalable barrier that a hardware barrier can provide. The BlueGene/L system [1] has several types of interconnection network among processors. One network, a three-dimensional torus, is used for regular data communication between processors. There is also a fast Ethernet network for accessing a file system and for network management. What is interesting is that there are two additional, special-purpose, networks: the collective network *and the* barrier network.

The collective network was designed to support global operations such as reduction, such as summing up elements of an array or matrix, taking their maximum, minimum, bitwise logical OR/AND/XOR, etc. The network is organized as a tree, in which each tree node is a node of processors. To sum up elements of an array, the summation starts at the leaves of the tree. A node collects the sum from its children nodes, accumulates the value to its own sum, and propagates it to its parent. Since children nodes can perform the summation in parallel, the total latency for such an operation scales as $\mathcal{O}(log\ p)$. The collective network also supports broadcast of data starting from the root to all nodes going down the tree. This can be done in parallel with regular data communication across the torus. Each link in the collective tree network has a target bandwidth of 2.8 Gb/s so it takes only about 16 cycles to transmit an 8-byte value (excluding the latency at the switch).

The barrier network is a separate network that supports global barrier. Instead of relying on a software barrier and cache coherence protocol, the barrier network works literally by sending signals along the wires in the network. The barrier network is organized as a tree. To perform a global barrier, a node collects signals from its children nodes, and propagate the signal up to its parent node, and this is repeated until the root node gets the signal. When the root node gets a signal, it knows that all nodes have arrived at the barrier. Next, it needs to release the nodes from the barrier. To achieve that, it sends a release signal down the tree to all nodes. The signals that are sent around are processed by special hardware logic at the barrier network controller of each node. Hence, they can propagate very fast. The round-trip latency of a barrier over 64K nodes is reported to take only 1.5 μs, corresponding to only a few hundred 700-MHz processor cycles, which is very fast compared to any software implementation (which easily takes that much time or more for just a few nodes).

8.3 Transactional Memory

In the context of parallel programming, transactional memory (TM) is designed to provide a higher level of programming abstraction that frees up programmers from dealing with lower level thread synchronization constructs such as locks. A code region enveloped as a transaction appears to execute fully or not at all (atomicity) without affected by interference from other threads (isolation). Earlier in Section 2.2.3, we have discussed an overview of transactional memory programming model, and in Section 4.4, we have discussed one application of transactional memory on linked data structure parallel programming. In this section, we will discuss architecture support to support transactional memory (TM) programming model.

There are three approaches for supporting TM. The first approach is to let software implement it while the hardware only provides the most primitive form of support in the form of atomic instructions – which are already in place to support other synchronization primitives. Such an approach is referred to as software transactional memory (STM). A particularly useful atomic instruction is the compare and swap (CAS) instruction. In an STM, a data structure (object) is wrapped by another object that contains a pointer that points to the original object. If an object needs to be modified atomically, it will be copied to a separate space and modified there. This modification is not visible to other threads until commit. After the modification is completed, it is committed by executing the CAS instruction to change the pointer so that it points to the new copy of the object. Thus, a single CAS can commit a large amount of changes to the data structure. Conflicts between concurrent modifications to the thread are detected by ensuring that all data that is read by the committing transaction (*read set*) has not been modified by another transaction (*write set*). STM comes as a library specific to particular data structures that the library provides. STM suffers from a significant overheads in the event of no contention due to the maintenance and bookkeeping of the object meta data. A second, related, approach, is providing hardware support to accelerate STM.

In this section, we will discuss the third approach, hardware transactional memory (HTM), where hardware directly provides transactions. We can think of the LL/SC pair as a primitive form of HTM. The lock implementation of LL/SC provides an illusion of atomicity, where any code between LL/SC appears to be executed completely or not at all. A hardware transaction provides the same illusion, but for an (almost) arbitrary region of code. The illusion of atomicity for a transaction requires several elements. First, it requires a mechanism to detect *conflicts*, i.e., conditions that may violate the illusion of atomicity. Second, if a conflict is detected, it requires a mechanism either to undo all effects the transaction has made. Alternatively, it requires a *buffer and commit* mechanism where changes made by the transaction are buffered and gated (not made visible to other threads) until commit time where the changes are made public. In the LL/SC lock, a conflict with other concurrent attempts of acquiring the lock is detected when the address of the lock variable stored in a linked register is cleared following a write attempt by another thread. Similarly, in a transaction, a conflict is detected when data that is read or written by the transaction (read and write set) intersects with data written by another thread (write set). In the LL/SC lock, no store instructions are allowed between the LL and SC, so as to enable the effects of the lock acquisition attempt to be undone easily. In addition, the SC instruction itself is conditional in that it will not be performed if a conflict has been detected. Unlike the LL/SC lock, a transaction must encapsulate (almost) all types of instructions, including store instructions. Since stores change values in memory, a more sophisticated mechanism is needed to buffer the changed/speculative values until commit time.

■ *Did you know?*

In Intel's HTM implementation (RTM = restricted transactional memory), a transaction is bookended by XBEGIN and XEND instructions. In AMD's HTM implementation (ASF = advanced synchronization facilities), a transaction bookended by SPECULATE and COMMIT instructions. On RTM, programmers can explicitly abort a transaction using XABORT.

As of the time of the writing of this book, transactions are only committed with "best effort", i.e., there is no guarantee that a transaction in RTM or ASF will eventually succeed. A transaction may abort even in the absence of conflicts, in multiple situations. Some instructions (e.g., CPUID) will by default abort a transaction. There are events that will also abort transactions, such as interrupts, I/O requests, (likely) page faults, etc. If the number of speculative values is higher than what the speculative buffer can hold, a transaction will also be aborted. For example, if the cache used for holding speculative values has a cache associativity of four, up to four blocks holding speculative values can be kept in any cache set. A fifth speculative value block will cause a speculative buffer overflow which aborts the transaction. There are mechanisms that can be added to allow the speculative buffer to overflow to the outer memory hierarchy safely and without needing a transaction to abort, but they are relatively costly. As a result, in current HTM, programmers are advised to provide non-transactional code that can be executed as a "Plan B" when a transaction repeatedly aborts.

Before going into the detail of the hardware mechanism, let us review the requirement for transactions to be executed atomically and in isolation from other threads. Recall the concept of serializability: *a parallel execution of a group of operations or primitives is serializable if there is some sequential execution of the operations or primitives that produce an identical result*. In this case, each transaction can be thought of as an operation or a primitive. Thus, transactions may be executed in parallel, but must produce the same outcome as if they are executed one at a time. Figure 8.2 illustrates serializability of transactions.

Figure 8.2(a) shows two transactions T1 and T2. Suppose that T1 writes 1 to x, reads from y, and writes 1 to z. This means that the write set for T1 includes x and z, while the read set for T1 includes y. Suppose that T2 reads from x, writes 2 to y, and writes 2 to z. This means that the write set for T2 includes y and z, while the read set for T2 includes x. A sequential execution of the transactions will either be T1 followed by T2 (shown in part (b)), or T2 followed by T1 (shown in part (c)). The values obtained by the reads are shown in bold italic font, and the final values of x, y, and z are shown at the bottom. These values show the only possible values that result from serializable execution of T1 and T2.

Now let us consider two parallel execution scenarios of T1 and T2, shown in parts (d) and (e). In part (d), T2's read of x sees a value of 1, and T1's read of y sees a value of 2. If we inspect the final values, they are (1,2,1), which agrees with part (c) where T2 executes prior to T1. However, the read from x is not the same in the two cases (0 vs. 1), which indicates non-serializable outcome has been produced in the execution scenario shown in part (d). The conflict in the two cases is the result of the intersection of a read from x from T2's read set and write to x from T1's write set. In part (e), T2's read of x sees a value of 1, and T1's read of y sees a value of 2. If we inspect the final values, they are (1,2,2), which agrees with part (b) where T1 executes prior to T2. However, the read from y is not the same in the two cases (0 vs. 2), which indicates non-serializable outcome has been produced in the execution scenario shown in part (d). The conflict in the two cases is the

result of the intersection of a read from y from T1's read set and write to y from T1's read set. This example illustrates that when two transactions have overlapping read vs. write sets, their parallel execution may produce non-serializable outcome.

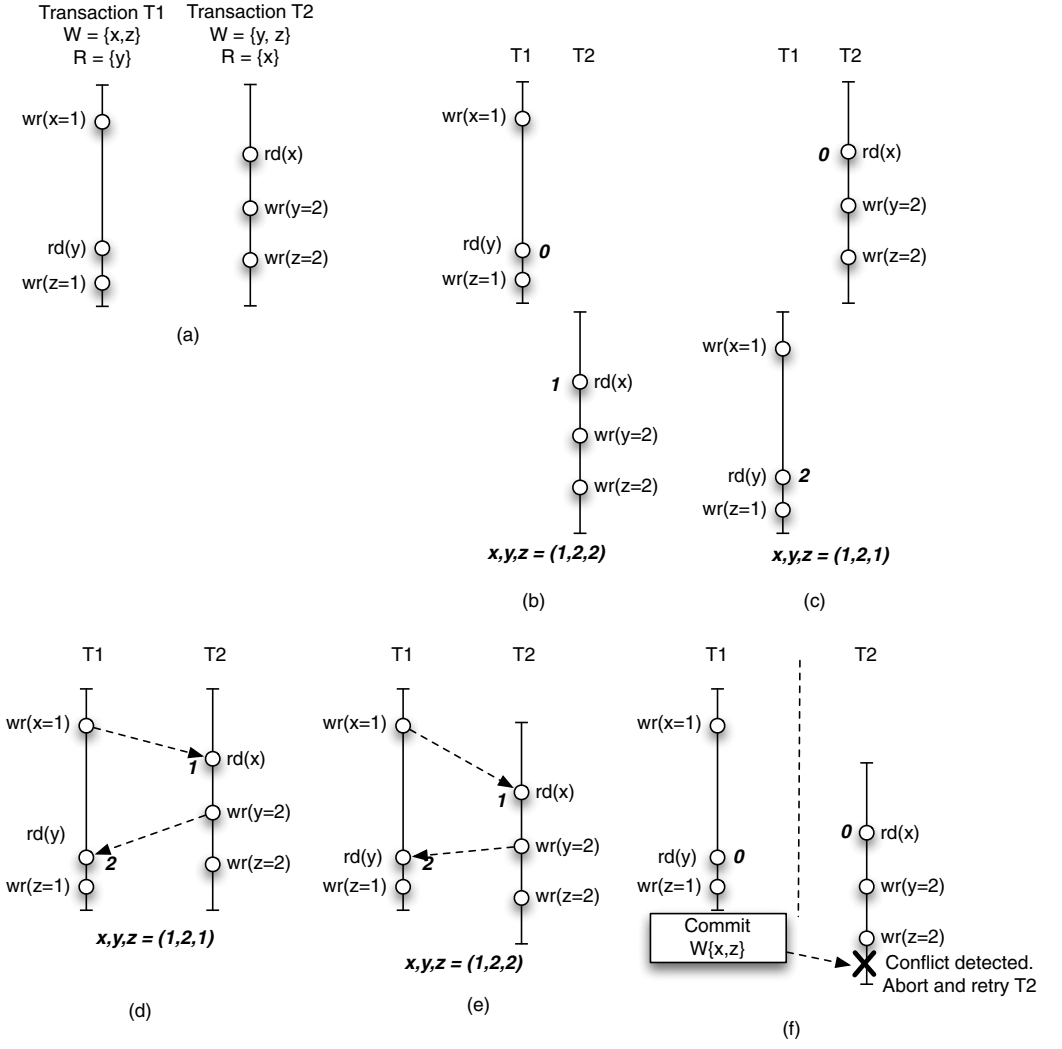

Figure 8.2: Illustrating serializability of transactions. Assume x, y, and z are initially 0.

In parts (d) and (e), we have assumed that the two transactions execute concurrently and propagate their values immediately upon writing new values to memory addresses. While this still allows detection of serializability conflicts, there is no easy way to abort a transaction and undo its effects as the writes have propagated to the memory hierarchy and other threads. In part (f), we assume a buffer and commit approach where the transactions are executed concurrently while we buffer their results to be committed at a later time. In this case, both T1 and T2 execute simultaneously, both assuming old values of x, y, z when they start execution. Conflicts are not detected until T1

commits its transaction where it publishes its write set $\{x,y\}$. At T1's commit, T2 finds out that T1's write set overlaps with its read (both containing x) and its write set (both containing z). Thus, a serializability conflict has been detected, and the right action here is to abort T2, discard its results (easy as they have been buffered and not propagated beyond the buffer), and retry it in the future.

Let us first discuss an approach where speculative values are buffered until commit time. One question is how and where these values should be buffered. Since buffered values should not become visible to other threads, the hardware must provide a space to keep these values without triggering the cache coherence to propagate writes. Several structures that are parts of the memory hierarchy can be used for buffering purposes. Examples of buffer structures include the store queue (used in Sun Rock), the L1 cache (used in Intel Haswell and AMD), and the L2 cache (used in IBM BlueGene/Q). Each of these buffer choices affect the transaction size and performance. The farther away the buffer is from the processor, the larger the capacity for buffering speculative values. For example, the store queue may hold up to a few hundred bytes, the L1 cache may hold a few tens of kilobytes, and the L2 cache may hold hundreds of kilobytes to a few megabytes. At this point, processor makers are not sure yet what maximum capacity they should provide for a transaction, as commercial programs are not yet written or ported massively with transactions.

One possible buffer to hold speculative values is a cache. A consequence of using the cache for this purpose is that tracking of speculative values is that data values will be tracked at a cache line granularity. This introduces a possibility of *false conflict*, where a transaction writing to a non-conflicting data is detected as a conflict with other data that is co-located on the same cache block. Another problem with using the cache for holding speculative values is that the transaction's size limit depends more on the associativity of the cache instead of the aggregate cache capacity. The reason is the transaction overflows the cache as soon as any single cache set holds as many speculative blocks as the cache associativity. For example, in the worst case, a 4-way associative cache may cause an abort after a transaction reads/writes to 5 cache blocks, if the blocks all map to a single cache set. There have been proposals to let speculative cache values to overflow to the main memory to provide an unbounded transaction size, but none of them is simple to implement. Furthermore, there is no proof yet that transactional memory can achieve or sustain high scalability when the transaction size is large.

Let us consider the implementation using a cache to hold speculative values. Each cache line is augmented with a *write bit* (a *read bit* is also added for tracking the read set). When transaction starts execution, any time it writes to a data item, the data is loaded into the cache (if it is not already in the cache) and its write bit is set. This way the block can be tracked as part of the write set of the transaction. In addition, the write bit also pins the block in the cache, making it non-evictable until commit time. If the transaction reads a data item, the data is is loaded into the cache (if it is not already in the cache) and its read bit is set. As with the write bit, the read bit has two roles: marking the read set of the transaction, and pinning the block in the cache until the transaction commits. If the transaction reads and writes a lot of data, there is a chance that the cache fills up and is unable to find a block to victimize. The typical and simple way to handle this is to abort the transaction. Such a policy carries a risk that there may be transactions that can never commit due to their size, and requires programmers to provide a "plan B" code that is invoked in case transactions fail to commit.

Transaction commits must be atomic, i.e., only one transaction commits at a time. For this, there needs to be a mechanism to arbitrate between potentially simultaneous attempts of transaction commits. A transaction commits by making sure that it has the ability to read from its read set and

write to its write set. First it has to determine if between the time the transaction starts until it wants to commit, whether there has been other transaction commits that have their write set overlapping with its own read and write set. If so, there has been a conflict and the transaction has to abort and restart. Once it is granted permission to commit, it can publish its write set. One way to publish the write set is by issuing invalidations to all blocks that are in its write set. The write set publication may cause other transactions to be aborted. Invalidating blocks that are in a transaction's write set also instantaneously propagate its writes so that they are visible to other threads. After it has published its write set, it can clear up the write bit and read bit and continue execution past the transaction.

If a transaction snoops another transaction's commit before it has a chance to commit, it has to compare the committing transaction's write set against its own read and write set. If they intersect, it has to abort. It aborts by going to the execution state of the processor prior to the start of the transaction. It has to restore the register state to that point, and invalidate all data in its read and write set.

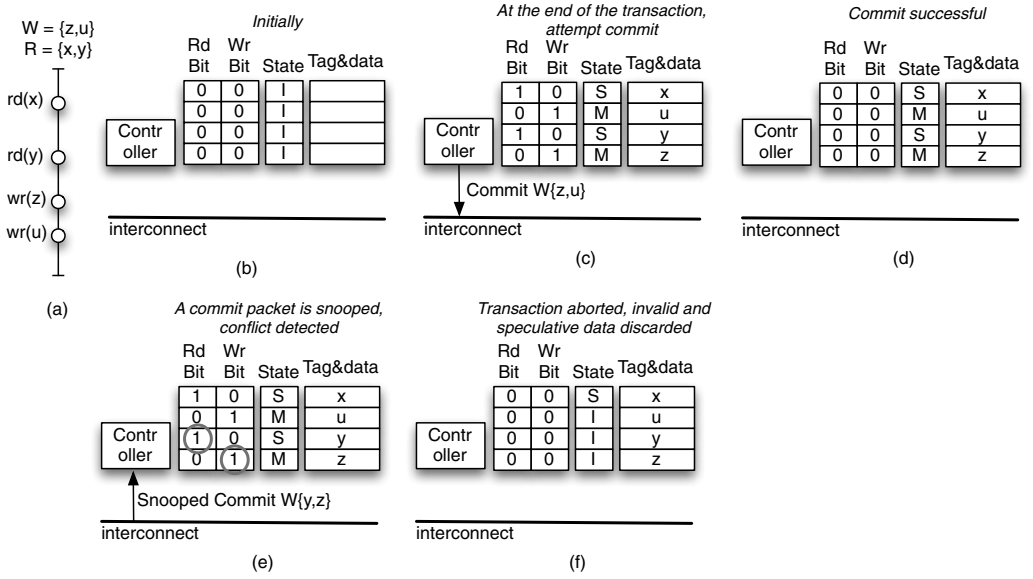

Figure 8.3: Illustration of transaction commit and abort. The transaction (a) and the initial cache state (b). Cache state before commit (c) and after successful commit (d). Cache state before commit upon snooping an external commit (e), and after transaction abort (f).

Figure 8.3 illustrates the scheme. Suppose we have a transaction shown in part (a) of the figure. The transaction reads from x and y, and writes to z and u. Part (b) shows the initial cache state, showing an empty cache with all read and write bits reset. Part (c) shows the cache when the transaction has finished execution, and is attempting to commit, but the commit has not succeeded yet. It shows that block x, y, z, and u are cached. The states of the block do not matter yet at the

moment as they are speculative. For illustration, we show x and y in a shared (S) state, and z and u in a modified (M) state. The read bits are set for blocks x and y and the write bit is set for blocks z and u. The commit request attempts to publish the transaction's write set by posting invalidations for z and u, in order to propagate the new value of blocks z and u, and notify other transactions of the write set of the commit. If invalidations to both blocks z and u are successful, then the commit can proceed by clearing/resetting the write and read bits in the cache. After the reset, regular cache coherence operations can be serviced to all data blocks that were involved in the transaction.

Parts (e) and (f) of Figure 8.3 show an alternative situation where before the transaction can commit, it snoops a successful external transaction commit. The external commit affects blocks y and z – which may come as one commit packet or as separate snooped invalidations to blocks y and z. Part (e) shows that a conflict is detected because the external commit's write set intersects with the transaction's read set (block y) and write set (block z). After the conflict is detected, the transaction has to be aborted. It is aborted by invalidating block y and z, and clearing all read and write bits, and retry the transaction. Block u was not involved in the conflict but holds speculative value, hence it has to be discarded and invalidated as well.

We have discussed one particular implementation of an HTM, especially one with the following features: write set published at commit time and speculative values buffered at the cache. It is not the only possible implementations. An alternative approach is to publish the write set early, every time a transaction writes to data. The latter is referred to *eager* conflict detection. With the eager policy, every write results in sending an invalidation message to other caches. Other transactions may abort as a result, and may miss on the block upon retries. However, the intervention requests will not be responded to until the transaction commits. The eager policy can potentially reduce useless work as conflicts can be detected early and transactions can be aborted and retried early. However, it can potentially cause too many aborts and retries. With regard to buffering speculative values, an alternative is to let the memory addresses to be updated with the new values. The old values are recorded in a log so that if the transaction is aborted, the old values can be restored. The approach is referred to as HTM with undo logs. An advantage of using logs is that commit is simple, because values are already propagated. A transaction simply needs to discard the undo log upon commit. A drawback of using logs is slower abort. If a transaction has to abort, it has to restore the old values from the log, one entry at a time. Furthermore, there is a serious drawback of using logs. Using logs requires a write to be changed into one read and two writes (one read of the old data, one write of the new data, and one write of the old data to the log) that must be performed atomically. If any one of them has occurred but a fault occurs before the others can be performed, the data or log can become inconsistent. The OS also has to be involved in allocating memory space for the log, which makes this scheme harder to implement.

■ *Did you know?*

To illustrate how a transaction is specified, the following code shows an implementation of compare and swap on two memory locations using AMD ASF [5]. The code reads mem1 and mem2 and compares them with register values held in registers RAX and RBX, respectively. If they are equal, then the swap is performed by writing register values in registers RDI and RSI to mem1 and mem2. To indicate the outcome of the swap, register RSX is reset (by XORing register RCX with itself) on swap, or leave it with a value of '1' otherwise. The code uses immediate retry as the recovery strategy. Other recovery strategies are possible.

```
1  DCAS:
2          MOV R8, RAX
3          MOV R9, RBX
4  retry:
5          SPECULATE ; Speculative region begins
6          JNZ retry ; Page fault, interrupt, or contention
7          MOV RCX, 1 ; Default result, overwritten on success
8          LOCK MOV RAX, [mem1] ; load value in mem1 into RAX
9          LOCK MOV RBX, [mem2] ; load value in mem2 into RBX
10         CMP R8, RAX
11         JNZ out       ; if R8 != RAX, jump to out
12         CMP R9, RBX
13         JNZ out       ; if R9 != RBX, jump to out
14         LOCK MOV [mem1], RDI ; store value in RDI to mem1
15         LOCK MOV [mem2], RSI ; store value in RSI to mem2
16         XOR RCX, RCX         ; Indicate swap was successful
17 out:
18         COMMIT ; End of speculative region
```

8.4 Exercises

Worked Problems

1. **Lock performance**. Consider a four-processor bus-based multiprocessor using the Illinois MESI protocol. Each processor executes a TTSL or a LL/SC lock to gain access to a null critical section. The initial condition is such that processor 1 has the lock and processors 2, 3, and 4 are spinning on their caches waiting for the lock to be released. Every processor gets the lock once and then exits the program.

   The codes for TTSL lock and unlock implementation:

```
lock: ld  R, L         // R = &L
      bnz R, lock     // if (R != 0) jump to lock
      t&s R, L        // R = &L; if (R == 0) L=1
      bnz R, lock     // if (R != 0) jump to lock
      ret

unlock: st L, #0      // write ``0'' to &L
        ret
```

   Thus, the lock primitive has only two memory transactions: BusRd generated by the `ld` instruction, and a BusRdX generated by the `t&s` instruction.

   The codes for `LL/SC` lock and unlock implementation:

```
lock: ll   R, L         // R = &L
      bnz  R, lock      // if (R != 0) jump to lock
      sc   L, #1        // L=1 conditionally
      beqz lock         // if SC fails, jump to lock
      ret

unlock: st L, #0      // write ``0'' to &L
        ret
```

   Thus, the lock primitive has only two memory transactions: BusRd generated by the `ld` instruction, and a BusRdX (or BusUpgr) generated by a successful `sc` instruction (and no bus transaction if `sc` fails).

   Considering only the bus transactions related to lock-unlock operations:

   (a) What is the least number of transactions executed to get from the initial to the final state for test-and-test&set and LL/SC?

   (b) What is the worst-case number of transactions for test-and-test&set and LL/SC?

   Answer:

   Best case for test-and-t&s lock: 7 bus transactions

Bus Trans.	Action	P1	P2	P3	P4	Comment
	Initial State	S	S	S	S	Initially, P1 holds the lock
1	st1	M	I	I	I	P1 releases lock
2	ld2	S	S	I	I	P2 read misses after invalidation
3	t&s2	I	M	I	I	P2 executes t&s and issues BusRdX
	st2	I	M	I	I	P2 releases lock
4	ld3	I	S	S	I	P3 read misses after invalidation
5	t&s3	I	I	M	I	P3 executes t&s and issues BusRdX
	st3	I	I	M	I	P3 releases lock
6	ld4	I	I	S	S	P4 read misses after invalidation
7	t&s4	I	I	I	M	P4 executes t&s and issues BusRdX
	st4	I	I	I	M	P4 releases lock

Best case for LL/SC lock: 7 bus transactions

Bus Trans.	Action	P1	P2	P3	P4	Comment
	Initial State	S	S	S	S	Initially, P1 holds the lock
1	st1	M	I	I	I	P1 releases lock
2	ll2	S	S	I	I	P2 read misses after invalidation
3	sc2	I	M	I	I	P2 executes a successful sc
	st2	I	M	I	I	P2 releases lock
4	ll3	I	S	S	I	P3 read misses after invalidation
5	sc3	I	I	M	I	P3 executes a successful sc
	st3	I	I	M	I	P3 releases lock
6	ll4	I	I	S	S	P4 read misses after invalidation
7	sc4	I	I	I	M	P4 executes a successful sc
	st4	I	I	I	M	P4 releases lock

Worst case for test-and-t&s lock: 15 bus transactions

Bus Trans.	Action	P1	P2	P3	P4	Comment
	Initial State	S	S	S	S	Initially, P1 holds the lock
1	st1	M	I	I	I	P1 releases lock
2	ld2	S	S	I	I	P2 read misses after invalidation
3	ld3	S	S	S	I	P3 read misses after invalidation
4	ld4	S	S	S	S	P4 read misses after invalidation
5	t&s2	I	M	I	I	P2 executes t&s and issues BusRdX
6	t&s3	I	I	M	I	P3 executes t&s and issues BusRdX
7	t&s4	I	I	I	M	P4 executes t&s and issues BusRdX
8	st2	I	M	I	I	P2 releases lock
9	ld3	I	S	S	I	P3 read misses after invalidation
10	ld4	I	S	S	S	P4 read misses after invalidation

11	t&s3	I	I	M	I	P3 executes t&s and issues BusRdX
12	t&s4	I	I	I	M	P4 executes t&s and issues BusRdX
13	st3	I	I	M	I	P3 releases lock
14	ld4	I	I	S	S	P4 read misses after invalidation
15	t&s4	I	I	I	M	P4 executes t&s and issues BusRdX
	st4	I	I	I	M	P4 releases lock

Worst case for LL/SC lock: 10 bus transactions

Bus Trans.	Action	P1	P2	P3	P4	Comment
	Initial State	S	S	S	S	Initially, P1 holds the lock
1	st1	M	I	I	I	P1 releases lock
2	ll2	S	S	I	I	P2 read misses after invalidation
3	ll3	S	S	S	I	P3 read misses after invalidation
4	ll4	S	S	S	S	P4 read misses after invalidation
5	sc2	I	M	I	I	P2 executes a successful sc
	sc3	I	M	I	I	P3's sc fails, no bus transaction generated
	sc4	I	M	I	I	P4's sc fails, no bus transaction generated
	st2	I	M	I	I	P2 releases lock
6	ll3	I	S	S	I	P3 read misses after invalidation
7	ll4	I	S	S	S	P4 read misses after invalidation
8	sc3	I	I	M	I	P3 executes a successful sc
	sc4	I	I	M	I	P4's sc fails, no bus transaction generated
	st3	I	I	M	I	P3 releases lock
9	ll4	I	I	S	S	P4 read misses after invalidation
10	sc4	I	I	I	M	P4 executes a successful sc
	st4	I	I	I	M	P4 releases lock

2. **Using LL/SC**. Use LL/SC instructions to construct other atomic operations listed, and show the resulting assembly code segments.

   (a) `fetch&no-op L` performs an atomic sequence of reading the value in the location `L` and storing it back into the same location `L`.

   (b) `fetch&inc L` performs an atomic sequence of reading the value in the location `L`, increment the value by one, and write the new value to `L`.

   (c) `atomic_exch R, L` performs an atomic sequence of swapping or exchanging the value held in register `R` and location `L`.

Answer:

(a)
```
fetch-noop: LL R, L          // R = mem[L]
            SC L, R          // mem[L] = R
```

(b) fetch-inc:
```
LL   R, L        // R = mem[L]
add  R, R, #1    // R = R + 1
SC   L, R        // mem[L] = R
bscfail R, fetch-inc // loop back if SC fails
```

(c) atomic-exch:
```
ll   R2, L     // R2 = mem[L]
sc   L, R      // mem[L] = R
bscfail R, atomic-exch // loop back if SC fails
mov  R, R2     // R = R2
```

Note that the `mov` instruction is purposedly placed after the sc, which is important to make sure that the sc is more likely to succeed. It is safe to do that because the value to be assigned to R is in the register R2, and hence can no longer be affected by an intervention or invalidation of another processor.

3. **Implementing locks**. Implement `lock()` and `unlock()` directly using the atomic exchange instruction. The instruction `atomic_exch R, L` performs an atomic sequence of swapping/exchanging the value held in register `R` and location `L`. Use the following convention: a value of "1" indicates that a lock is currently held by a process, and a value of "0" indicates that a lock is free. Your implementation should not repeatedly generate bus traffic when a lock is continuously held by another process.

   Answer:

```
lock: mov R, #1          // R = 1
loop: ld R2, L           // R2 = mem[L]
      bnz R2, loop       // Lock not free, loop back
      atomic_exch R, L // exchange R with mem[L]
      bnz R, loop        // lock attempt fails, loop back
      ret                // lock successfully acquired, return

unlock: st L, #0         // release the lock
        ret
```

4. **Barrier implementation.** A proposed solution for implementing the barrier is the following:

```
BARRIER (Var B: BarVariable, N: integer)
{
  if (fetch&add(B,1) == N-1)
    B = 0;
  else
    while (B != 0) {};   // spin until B is zero
}
```

Describe a correctness problem with the barrier implementation. Then, rewrite the code for `BARRIER()` in a way that avoids the correctness problem.

Answer:

The correctness problem occurs when all but last thread have arrived at the barrier spinning on the while loop. Then the last thread arrives and sets B to 0. The last thread may then continue to the next barrier with the same name, and immediately increments B. Other threads that are

about to come out of the barrier have their cached copies of B invalidated, and reload them to find out that the value of B is no longer 0, and stay in the barrier.

The implementation can be corrected by alternating between spinning for 0 and for N-1, i.e.

```
BARRIER (Var B: BarVariable, N: integer)
{
  static turn = 0;

  if (turn == 0) {
    if (fetch&add(B,1) == N-1)
      B = 0;
    else
      while (B != 0) {};   // spin until B is zero
    turn = 1;
  }
  else {
    if (fetch&add(B,-1) == 1)
      B = N;
    else
      while (B != N) {};   // spin until B is zero
    turn = 0;
  }

}
```

Homework Problems

1. **Lock performance**. Consider a three-processor bus-based multiprocessor using MESI protocol. Each processor executes a test-and-test&set or an LL/SC lock to gain access to a null critical section. Consider the following sequence of events:

   - Initially: P1 holds the lock
   - P2 and P3 read the lock
   - P1 releases the lock
   - P2 and P3 read the lock
   - P2 acquires the lock successfully
   - P3 attempts to acquire the lock but is unsuccessful
   - P2 releases the lock
   - P3 reads the lock
   - P3 acquires the lock successfully
   - P1 and P2 read the lock
   - P3 releases the lock

   Considering only the bus transactions related to lock-unlock operations:

(a) Show the states of each cache for the sequence assuming the test-and-test&set lock implementation. Use the following template. Bus transactions corresponding to the first three steps are shown.

Bus Trans.	Action	P1	P2	P3	Comment
-	Initially	M	I	I	Initially, P1 holds the lock
1 (BusRd)	`ld2`	S	S	I	P2 read misses on the lock
2 (BusRd)	`ld3`	S	S	S	P3 read misses on the lock
3 (BusUpgr)	`st1`	M	I	I	P1 releases the lock
4	and so on ...				

(b) Show the states of each cache for the sequence assuming the LL/SC lock implementation. Use the following template. Bus transactions corresponding to the first three steps are shown.

Bus Trans.	Action	P1	P2	P3	Comment
-	Initially	M	I	I	Initially, P1 holds the lock
1 (BusRd)	`ll2`	S	S	I	P2 read misses on the lock
2 (BusRd)	`ll3`	S	S	S	P3 read misses on the lock
3 (BusUpgr)	`st1`	M	I	I	P1 releases the lock
4	and so on ...				

2. **Lock performance**. Consider a four-processor bus-based multiprocessor TTSL lock implementations shown in the book (discussed in lectures). Suppose the cache coherence protocol is MOESI, and we have the following events:

   - Initially, P1 holds the lock (and lock variable is cached in modified state)
   - P2, P3, and P4 reads the lock variable in that order
   - P1 releases the lock
   - P2, P3, and P4 reads the lock variable in that order
   - P4 acquires the lock successfully
   - P2 and P3 attempt but fail to acquire the lock in that order
   - P3 reads the lock variable
   - P4 releases the lock
   - P2 reads the lock variable
   - P2 acquires the lock successfully
   - P2 releases the lock

   Show for each event what bus transaction is generated (ignore Flush and FlushOpt for brevity), what instruction it corresponds to, and the resulting states in the cache.

3. **Lock performance**. Repeat homework problem (2) with LL/SC lock implementation.

4. **Lock performance**. Repeat homework problem (2) with Dragon protocol.

5. **Lock performance**. Repeat homework problem (2) with LL/SC lock implementation on MESI protocol.

6. **Using LL/SC**. Use LL/SC instructions to construct an atomic compare and swap instruction "CAS R1, R2, L" which tests whether data in memory location L is equal to that in R1. If they are equal, write the value in R2 to L, and copy R1 to R2. Otherwise, nothing is done and CAS returns. For example, if initially R1=5, R2=10, L=5, after CAS we have R1=5, R2=5, and L=10. If initially R1=7, R2=10, L=5, nothing changes after CAS. Show your answer in assembly code, and annotate what each instruction does. Keep as few instructions between LL and SC as possible.

7. **Lock implementation**. Use "CAS R1, R2, L" from problem (6) to construct a lock primitive. Lock(Location L) and Unlock(Location L). Keep the implementation short and simple, and avoid generating unnecessary bus transactions when the lock is being held by a thread.

8. **Read and write lock**. Show the machine code for implementing a read and write lock using the LL/SC primitives.

9. **Combining tree barrier**. Write the pseudo code for a combining tree implementation, assuming that the tree is binary (each node receives barrier completion signals from two "children" nodes, except the leaf nodes).