Synchronization Lecture Notes

BRIAN RAILING

With multithreaded code, threads are sharing memory and state with each other. Cache coherence ensures that each thread sees writes by other threads. But while coherence will further ensure that two writes will be ordered with respect to each, it does not ensure that these values are appropriate and meaningful. For example, multiple threads are incrementing a common variable: `x++`. Assuming `x` is a memory location, this requires three operations: load the value, increment the value, and store the value. So while coherence ensures the load observes a prior store, it does not guarantee that the operations will not interleave such that two threads both load the value, increment, and then each store.

To guarantee certain orderings of multiple operations, we must add additional constructs to the program, synchronization. These constructs ensure certain orderings or properties, such as only one thread may execute a sequence of operations concurrently. Collectively, we can think of the right to do perform these operations as requiring permissions, which must be requested and then granted. The managing entity of the permissions is most commonly termed, a lock.

Throughout this chapter, when some form of synchronization is required in the code examples, it will be termed *lock*. The lock may be any one of a variety of possible implementations, such as mutexes, spinlocks, or semaphores.

1 GUARANTEES

Must have: Mutual exclusion

Progress - gives a guarantee that a thread will acquire the critical section based on the actions of the waiting threads. Progress does not guarantee bounded waiting, as progress only requires that a thread proceeds.

Nice to have: Bounded waiting - the time for a thread to wait can be bounded basic on the number of requesters. But if the units are in requests, then a bounded system does not guarantee that progress is made.

2 MODEL

There are three phases to a synchronization operation: acquire, hold, and release. In the acquire phase, a thread executes one or more operations to ensure that it has the requisite permissions with respect to the other threads operating in the system. These operations themselves may require two phases: waiting and acquisition. In the hold phase, the thread performs the desired operations while holding the appropriate permissions to ensure correctness. And finally, the thread releases the permissions, such that the same or a different thread can acquire them at a later point in execution. Each lock implementation will affect the average time required to acquire and release the permissions; however, under high load the acquire time will converge on a multiple of the average hold time.

```
word test_and_set(*addr)
{
  // the following operations are performed atomically
  old = *addr;
  *addr = 1;
  return old;
}
```

Fig. 1. Test-and-set

We can consider two cases for a thread trying to acquire a lock (i.e., a set of permissions). In the first case, the permissions are available and the thread incurs the minimum expected overheads[1] to acquire / release the permissions. In the second case, the permissions are not available. The acquire phase is now extended by some time based on the thread with the permissions holding time. If the requester is the only thread waiting, we should expect this wait time to be half of the average hold time. With a second waiting thread, the waiting time extends from half plus the time for the next acquisition to complete an entire hold phase. And thus as the number of waiting threads increases (i.e., higher contention), the waiting time scales as multiples of the average hold time. However, we shall find that the lock implementation can cause the acquisition performance to deviate from this ideal.

3 BASIC DESIGN

For our basic design of a synchronization mechanism, we will use a common architectural primitive, test-and-set, see Figure 1. Being an architectural operation, the component operations can be performed atomically, meaning that no other thread can observe / effect the intermediate state of the operation. Thus, if many threads call test_and_set on a location that is initially 0, only one thread can possibly receive the return value of 0, while all other threads receive 1. This design gives a clear correspondence to implementing a lock. A thread trying to acquire a lock enters a loop continually calling test_and_set on the lock address until the return value is 0, indicating that it has set the lock's state. This loop is termed a spinloop.

To release the lock, we only need to restore the lock to its initial state, 0. While architectures usually do not require all operations on an address to be atomic, there is generally a restriction that if the accessed memory locations partially overlap, then this behavior is undefined.

If we return to considering cache coherence, how is an atomic operation treated by the processor. Is it a read, then optionally a write? That is how we would expect the execution when the operations are treated independently; however, as the architecture determines the request type and interprets the machine code, atomic operations are therefore almost exclusively performed when the memory location is in the M (modified) state, even if the operation may or may not modify the memory state. At one time, processor architectures would mandate the presence of a #LOCK line as part of the bus and that this line would be asserted during any atomic operation, thus locking the bus and ensuring that no other memory operations could take place (such as, Section **??**).

4 AVOIDING COHERENCE

In the basic lock designs, the releasing thread has no higher priority on the cache line than any thread attempting to acquire it. Given that an atomic operation effectively requires the cache line to have the modified permission, each

[1] Modulo the impact of caching, branch prediction, et cetera.

```
acquire_lock(*lock)
{
  while (test_and_set(lock) == 1)
    while (*lock == 1);
}
```

Fig. 2. Extending to Test-and-Test-and-set

```
acquire_lock(*lock)
{
  while (1)
    while (*lock == 1);
    if (test_and_set(lock) == 0) return;

}
```

Fig. 3. Refactored Test-and-Test-and-set

attempt to acquire steals the line away from the previous attempt. Into this queue of attempts, the releasing thread waits until it can signal the lock. As this coherence traffic is both directly wasteful, in that it generates no useful work, and further that the traffic is delaying the release path, we should update the basic lock to avoid these operations unless there is a chance of being successful. This change is a test-and-test-and-set lock, see Figure 2.

Each thread still attempts the atomic test-and-set operation as before, but now rather than just spinning, it spins on the "test" operation. These operations only read the state of the lock and verify that it is still held. Being reads, the line can be shared with other waiting threads and does not generate coherence traffic.

This code is optimistic, in that it assumes that the lock is available and thereby first attempts to acquire the lock. In contrast, if the lock is generally expected to have waiters, then the code can be refactored to Figure 3. In this alternative, the lock's state is first read before any attempt to acquire is made. This code can be slightly slower for an uncontended acquire, but can save significant coherence invalidations when there are many threads waiting.

Now in these designs, the waiting threads will all be in the same state, spinning while reading the lock state. On release, every waiter's cacheline is invalidated and the lock's state set to 0 (unlocked). At this time, each thread will attempt to read an invalid cacheline, incurring a miss and sending a coherence request for the line. As each cache is populated with the lock, the thread observes that the lock is not locked and exits its spin loop, then attempting to acquire the lock using test_and_set. However, this requires a new coherence request and the requesting thread's coherence operation is queued, while the prior reads from the other threads are serviced. The threads are guaranteed to make progress with respect to their memory requests, and therefore each of these waiting threads will read the new lock state of 0 and exit their spinloops. Thus the threads will all be making exclusive coherence requests in their attempt to modify the lock.

We are seeing then a stampede of threads from their quiescent reading spinloop to all attempting to update the line, where on the invalidation, each thread reissues the GetRd and then seeing the state change will issue a GetRdX for that line. Being queued, we will generally expect the threads to proceed together from one phase to the next. One thread will succeed, while the remainder will fail and again have to fall back to the read loop, thereby each requiring an additional GetRd bus request. So on release and for N waiting threads, there will be 3*N coherence requests.

As most threads are not making progress, why not have them check less often or back off its frequency of reading the lock? Ideally, a thread would only check the lock's state just after the lock has been released, before this time the

lock is not available and after this time adds unnecessary delay. However, the question is then how would a thread know when to check. [2]

There are variety of approaches to backing off and finding the right time to wait, let's term these approaches: Linear, exponential, and historic backoff approaches. In each backoff approach, the waiting time ranges between two bounds: a small number of cycles (say 1) and a small multiple of the context switch time. Beyond this multiple, some implementations fall back to a OS-scheduled lock, whereby the thread is descheduled and the hardware context made available for other threads to run. Otherwise, the thread will continue spinning, primarily counting the time before it checks the lock again.

In linear backoff, the thread will check after 100, then 200, 300, et cetera cycles. While an exponential backoff will do so after 100, 200, then 400 cycles. The historic is a modification whereby the lock records the average (moving, weighted, or otherwise) of the lock acquire times and uses this as initial guidance into how much backoff should a thread use. Recall, when attempting to acquire the lock that is already held, then we should expect to wait half of the average hold time before the lock is released. The backoff algorithms are trying to efficiently find this average hold time, while also taking into account that we should expect half of these acquire attempts will be ideally waiting for less than this amount of time.

When backoff is implemented with these spinlock designs, ideally the 2*N coherence requests is reduced to a constant term, mitigating the impact of a significant number of waiters.

5 OTHER ATOMIC PRIMITIVES

To extend our spinlock designs and later provide additional complex atomic operations (see Section 9.2), we need several additional atomic primitives. We'll discuss three primitives here, with the either of the later two, we can build all atomic primitives. Depending on the specific architecture, different atomic primitives are provided in the instruction set.

5.1 What makes it atomic

Hardware provides atomic instructions that support updating memory based on the prior state. `test_and_set` was one example instruction, but architectures can implement many others based on the common operations that programs wish to execute.

5.2 Atomic Fetch and X

Many of the simple integer operations can be directly performed atomically without a separate lock. Incrementing a variable, `x++`, can be prefixed to be atomic. And processors may extend to more generally, `x += y` or `x &= y` et cetera. Given the load placed on the memory during these operations, they are often constrained to single-cycle operations to avoid locking the bus for the tens of cycles required for a division.

5.3 Compare Exchange

Until now, all of the atomic operations always executed completely, regardless of the current state of the system. With compare exchange (CMPXCHG), the processor's execution will vary depending on the state of the system, while maintaining the appearance that the instruction executed atomically. The basic idea for CMPXCHG is that the program only wants to modify the memory state if it contains a specific value, otherwise the operation should be unsuccessful.

[2] This is a possible area of research whereby the architecture could provide an instruction to check the coherence state of a cache line and respond accordingly.

```
// All of these instructions are performed atomically
CMPXCHG(*loc, old, new)
{
  ret = *loc;
  if (ret == old) *loc = new;
  return ret;
}
```

Fig. 4. Pseudo-code for Atomic Compare Exchange (CMPXCHG)

In Figure 4, we see this pseudo-code implementation for CMPXCHG. The caller provides a memory location, the old / expected value in that location, and the new value to store if the old value is found currently in that location. This *if* is what gives this instruction its power.

5.4 Load-linked / Store-conditional

The most general of these operations is the pair of instructions, load-linked (LL) and store-conditional (SC). These operations are a simple form of Transactional Memory (see Section 11). On executing a LL instruction, a memory location is read into a register. The code can now execute other, non-memory instructions. Finally, when the SC executes to store a value back to the memory location read by LL, the processor checks whether that memory location has been accessed by any other hardware context, if not, then the store is successful, otherwise, the store fails and the appropriate condition codes are set. In some hardware designs, this constraint can be more conservative.

6 FAIRNESS IN LOCKING

The problem with the locking approaches thus far is that while providing good average access time, there can be significant discrepancies in time due to a lack of fairness, otherwise termed bounded waiting. As without this guarantee, there is no assurance that a waiter may ever acquire the lock in a continuously used system. Instead, we can change the locks to enforce this property either through a static or dynamic ordering, and to do so we need to use the primitives from the previous section (Section 5).

6.1 Ticket locks

The ticket lock is designed like a deli counter. Each thread arrives and takes a 'ticket', which is a number indicating the thread's order, as acquiring a ticket can be a single operation that always succeeds, it implements the progress property. The lock also maintains a number indicating which ticket currently holds the lock. Thus as we see in Figure 5, when locking, the thread waits until its ticket is the lock holder, and on unlocking the thread hands off the lock to the next ticket. However, each unlock operation invalidates all of the waiters, which were all reading the serving number. And as each continues to wait, they will need to reread the cacheline. Thus the traffic is O(waiters) per acquisition.

Given the invalidations created by the release, lower active contention is desirable. However, this design is simple and space minimal.

6.2 Array-based locks

To avoid the high invalidation count with the ticket locks, we can instead create an array of waiting flags, each on its own cache line. When a thread arrives, it acquires a flag element slot and waits for it to be signaled. Being on its own

```
Lock(*lock)
{
  int ticket = atomic_increment(lock->next_ticket);
  while (ticket != lock->serving);
}

Unlock(*lock)
{
  lock->serving++;
}
```

Fig. 5. Ticket-based Lock Implementation

```
acquire_lock(*my_lock, *global_lock)
{
  my_lock->next = NULL;
  my_lock->state = WAIT;
  prev = my_lock;
  ATOMIC_SWAP(global_lock, prev);
  if (prev == NULL) return;
  prev->next = my_lock;
  while (my_lock->state == WAIT) ;
}
```

Fig. 6. Acquire Queued (MCS) Lock

cache line, its flag will remain cached until the lock is released from the immediately previous acquisition, which will set the flag of the next waiter. Thus each acquisition requires a request to receive its slot and then transfer requires two more coherence messages.

The limitation of this approach is that the memory used is allocated at compile-time or lock creation time. This bounds the maximum number of threads that can wait on the lock, and furthermore commit excess memory to the lock during low contention scenarios.

6.3 Queued spinlocks

Queued spinlocks[3] provide both the guarantees of previous lock implementations, while also dynamically scaling the resources required. This lock implementation creates a queue, via a linked list, of waiting threads, where each thread has its lock variable to spin on. This last feature is also important as this avoids many of the earlier coherence and contention problems.

In Figure 6, the acquire lock design takes in two parameters: a local lock instance, often allocated on the stack, and the global lock state. This global state holds two possible values: NULL indicating that the lock is not acquired, and a non-NULL value pointing to the last thread enqueued (which could be currently holding the lock). The local lock instance holds two pieces of data: whether this thread has the lock and where to find the next waiting thread. Thus, the waiting threads form a linked list, with the global lock state pointing to the tail of this list and the thread at the head of the list currently holding the lock.

[3]While a variety of implementations exist, I will discuss the MCS design.

```
release_lock(*my_lock, *global_lock)
{
  do {
    if (my_lock->next == NULL) {
      x = CMPXCHG(global_lock, my_lock, NULL);
      if (x == my_lock) return;
    }
    else
    {
      my_lock->next->state = ACQUIRE;
      return;
    }
  } while (1);
}
```

Fig. 7. Release Queued (MCS) Lock

A queued spinlock requires five coherence requests to complete. Let's work through each. First is the access to `global_lock`. When `global_lock` returns the tail of the queue, this lock node is then accessed, which is in the previous thread's cache. This access results in the releasing thread also incurring a miss to access its `my_lock` and then a second miss to update the state for the next lock. After this update, the acquiring thread's lock state is invalidated and reaccessed to incur the fifth coherence miss.

The major reason that queue-based locks are not more commonly used is that they require a different interface than the other locks, as the local waiting node needs to be allocated dynamically (typically on the stack). This requirement prevents this lock from being dropped in as a replacement to the other implementations.

6.4 Drawbacks

There are two major drawbacks to these designs. First, each implementation requires more operations to perform and therefore increase the uncontended acquire time. For example, the test-and-set design requires a single operation to acquire, while the queued lock requires several, although both still require a single atomic operation.

The second drawback is that following a strict notion of fairness, the lock acquiring order must wait for the next thread, even if that thread is not currently scheduled to run. This scenario is most common when the number of threads exceeds the number of hardware contexts, and the operating system is forced to make scheduling decisions about which thread should be running. Given that a waiting thread is effectively idle (i.e., producing no work), it would be reasonable to be descheduled, except if the lock is "fair" and will be given to it regardless of its runnable status. Thus, with the number of threads in excess of the hardware contexts, it becomes increasingly likely that a given thread will not be running and therefore not use the "fair" lock when provided. The ultimate effect of this design is that the non-running thread will "hold" the lock for milliseconds until it runs again, and this time is significantly larger than normal hold time. This then skews the acquire times, as they are ideally the average number of waiters times the average hold time.

It has been proposed that by adding timestamps to the queued spinlock design, such a waiting thread can periodically check the waiter ahead of it and steal the provided lock. Or on release, the releasing thread can inspect the next thread's wait status and determine whether it is running and therefore whether it should be provided the lock, or if the lock should instead be provided directly to the next waiting thread following it. I suspect the later is more likely.

```
struct reader_writer {
  lock reader, writer;
  int reader_count;
};
```

Fig. 8. Reader-writer State

7 READER-WRITER LOCKS

Just as coherence supports multiple hardware contexts sharing a cache line and the exclusivity to modify that cache line, so can we design synchronization to ensure similar sharing can take place safely while supporting the capability to modify the data. To achieve this, we extend the use of a synchronization construct to protect special reader-writer data requiring exclusion, rather than the program's data itself, see Figure 8.

Readers can enter by acquiring the reader lock, incrementing the count and dropping the lock. On release, the reader operates similarly. If the reader count increments from 0 to 1, then the writer lock is acquired. And on release, if the reader count drops to zero, the writer lock is released. For writers, the writer lock is the sole element of mutual exclusion acquired and released appropriately.

7.1 Reader or Writer Bias

Reader-writer locks can be designed to be ordered or to bias toward either readers or writers. When the lock is in "reader mode", it is providing the greater concurrent access to the data, versus the "writer mode" which is provides exclusive access to the data.

Consider the case that 1 thread is currently reading the data and 1 thread is waiting to write, when another thread arrives requesting to read the data. This newest reader could access the data concurrently improving performance, but in doing so the lock would violate fairness and further risk starvation (if readers continually arrive).

Now consider the case where 1 thread is currently writing the data and 1 thread is waiting to read the data, when another thread arrives requesting permission to write to the data.

Effectively, the bias should be made toward the infrequent requester. That said, if writing is sufficiently common, then one should just use mutual exclusion instead, as the reader-writer design incurs additional overhead for the rarely occurring reader.

7.2 Scalable Locking

The reader-lock designs so far still have a single point of serialization. However, this state can be replicated into multiple instances such that a reading thread only needs to register itself at a single instance, while a writing thread must now acquire exclusivity from every instance. This approach relies on there being a inexpensive way to determine the unique lookup, such as thread or CPU identifier.

8 BARRIERS

Contrasting with the synchronization approaches thus far, barriers are designed to handle a different problem: how to ensure that N threads have arrived at a point P in the program before any of those threads can execute instruction P+1. The basic design requires a count and a flag. As each thread arrives, it atomically increments the count (probably while

```
barrier_wait(*bar)
{
  lock(bar->lock);
  old_flag = bar->flag;
  bar->count++;
  if (bar->count == N) {bar->flag = !old_flag; bar->count = 0;}
  unlock(bar->lock);

  while(bar->flag == old_flag) ;
}
```

Fig. 9. Pseudo-code Flip-flop Barrier Implementation

holding a lock) and if the count is equal to the expected number of threads, this last one then signals the waiters using the flag.

Ideally, the barrier would be reused, for example as the last operation in a loop body. In our simple design, one thread is released by the barrier continues executing. It arrives at the barrier, increments the counter, and then waits on the flag which is already set so it keeps running. The final thread to arrive, could reset the counter, such that on the next iteration the first thread clears the flag; however, there is no guarantee that all threads have exited their barrier spinloop when the flag is then reset. Ignoring the option of making all barriers be one off, we have two possible solutions.

First, the barrier could be designed such that one specific thread is the designated signaler, as it is in OpenMP, or that the last thread will do so. Then the barrier has two phases, the first gathers the threads and holds them waiting. When all have been gathered, the signaling thread releases the gathered threads. This signaling thread then must wait on the second phase until all have been released.

8.1 Sense Reversal

Second, the meaning of the barrier's flag can be changed. If it is treated similar to a electronic signal, the notification comes from it switching state rather than being set. In this design, each arriving thread records the current flag state before incrementing the count. Then after the increment, it waits for the flag state to have changed, which comes from the last arriving thread signaling the barrier. This extension gives us the implementation in Figure 9.

8.2 Barrier Topology

In the above barrier design, the common state is updated by every arriving thread. And given that the state requires a lock, this forces the span to by $O(P)$. It is possible to do better. Let's construct a tree of barriers, which could be a branching factor of 2 but any small constant will work. Each barrier has its two phases: gather and release. When all of the assigned threads arrive at a specific barrier node, the last thread then arrives at the parent node to this barrier. This sequence continues until the last thread arrives and traverses all the way to the root of the tree, at which point the barrier enters the release phase and threads release the threads waiting at each barrier node. While the total work has increased, the span has been reduced as the state updates are no longer serialized.

Practically, this serialization may not be an issue in most programs. If the interval between threads arriving at the barrier exceeds the expected time required to update the barrier state, then the barrier state would primarily be uncontended and the threads would not have to wait. Put another way, if the time between the first and last thread

arriving at the barrier exceeds $O(P) * T(update)$, then the contention free time exceeds the serialized time. That said, as thread count or memory transfer time increases, the need for a decentralized barrier will also increase.

9 FINE-GRAINED SYNCHRONIZATION

The synchronization thus far has been coarse-grained, meaning that each construct covers a large set of operations; however, just as we saw with reader-writer locks, there are opportunities to increase the supported concurrency without sacrificing correctness. We will explore how synchronization can be reduced in scope, and in so doing, some operations can be executed concurrently rather than serialized by the synchronization.

9.1 Data structure locking

Program data is often stored in dynamic structures, such as trees or linked lists. The default strategy for safe, concurrent accesses is to place a single lock around the entire data structure. The simplification afforded by this approach makes it reasonable for many uses, and particularly for data structures that have infrequent accesses and are therefore unlikely to need to handle concurrent accesses.

If we consider inserting elements into a sorted linked list, the operations are independent of each other and could be completed any order or even concurrently. The only restriction would be safe modification of an element's `next` pointer. And similarly for deletion. We can accomplish this by adding a lock to each element. So if we are inserting C into `A->B->D`, the code would `lock(A)`, then `lock(B)`, `unlock(A)`, and `lock(D)`. Then while B and D are locked, C can be safely inserted between them. Similar modifications can be made to other pointer-based data structures.

However, this is not without costs. Each element requires additional space to maintain its lock. Furthermore, these locks now introduce time overhead with their acquisition and release on every operation on the data structure. This overhead is rather expensive such that a high rate of concurrent access is needed before the overall program benefits.

9.2 Atomic Operations versus Locks

Many forms of synchronization discussed so far have relied on atomic operations provided by the architecture. Some, such as `test_and_set`, have a clear use in lock implementations, while others, such as `compare_and_swap`, are more generalizable. Extending from these operations, architectures may implement many more atomic operations, such as `fetch_and_add`. Against this backdrop of operations, we should revisit our use of synchronization in solving many problems.

> As an aside, this is one style of performance optimization that can be hard to explain, such as integrating an exclusivity bit with an atomic counter. In the original design, the system would acquire a lock, check a series of conditions, including incrementing a count of items, and then release the lock. With most of the conditions being errors (and thus infrequently set), much of this synchronized time was maddeningly unnecessary. I proposed creating a fast lookup design, whereby the conditions (if *safe*) and the count of items would be in a single field. Using `compare_and_swap`, a thread could verify the conditions and register its operation via the count. Separately on every error path, besides setting the specific fail condition, the execution would also clear the *safe* condition from the fast lookup field. Threads would then fallback to the pre-existing lock-based code until one found all conditions to be *safe* again. I thought it elegant, the others thought it dangerous.

```
atomic_OP(*addr, val)
{
  do {
    old = *addr;
    new = old OP val;
  } while (compare_and_swap(addr, old, new) != old);
  return old;
}
```

Fig. 10. Atomic Read-modify-write Loop

Our initial example in this chapter saw multiple threads attempted to update a common counter, `x++`. Simple guidance would ask programmer's to add a synchronization construct to protect this counter. And should a similar field be part of a dynamic structure, then each element may require its own construct. However, we have a simpler, and more importantly, faster way to solve this concurrency problem by utilizing the underlying atomic operations. For example, `x++` can be replaced with `fetch_and_add(x, 1)`, where the later is guaranteed atomicity. Even if the architecture does not directly provide a single instruction, the compiler (or programmer) can construct a read-modify-write loop that will perform the operation atomically, see Figure 10. OpenMP, via the `#pragma omp atomic`, and C/C++1x standards, via the atomic datatypes as prefixed with `_Atomic`, provide access to these operations without requiring the programmer to discern which operations are specifically available from the architecture.

9.2.1 Atomics as bitpacking. Locks often waste space. Consider the following:

```
atomic_add(int* loc, int val)
{
    lock()
    *loc += val
    unlock()
}
```

Looks fine, right? Consider that the lock is 4 bytes of space, of which 1 bit indicates whether it is acquired. Two compactions are possible. First, the lock could be integrated into the value itself. This would reduce the range for the value, as one bit is now dedicated to holding the lock state. Or second, many architectures (like x86) support the above operation as a single instruction. So using the instruction, removes the need for a lock bit / word entirely.

Why don't we always replace the locks with atomic instructions? Two reasons: first, if the locked section is ever more complex, then the atomic sequence becomes vastly more complex, as only the simplest of operations (add, subtract, or, and, etc) are supported with atomicity. Second, the determination of the appropriate transformations in the compiler is exceedingly complex and may not be solvable in the general case. Since the compiler wouldn't be able to do this generally, it looks instead to the programmer to introduce this change.

10 LOCK-FREE SYNCHRONIZATION

Until now, our synchronization has a terrible flaw. If the lock holder is context switched out, no progress can be made. Often we assume this is an infrequent event, which is valid but it does happen. Some scenarios are more concerned by worse case performance than average. Into this space, we can do away with locks. We cannot remove atomic operations,

but we can guarantee that regardless of the order of any context switches, as long as one thread is running, the code will make progress. Do note that using lock-free code provides this guarantee, but at a cost in both implementation complexity and best case performance.

10.1 ABA

Many lock-free implementations are susceptible to having limited ability to detect changes to the data structure. In using compare-and-swap, the architecture is asked to operate based on whether the expected value is still present at that memory location. For commutable operations, such as atomic adds, this behavior matches the programmer's model. In lock-free data structures, finding the expected value present does not explicitly indicate that the data structure is unchanged.

For example, thread 1 is ready to pop $A \rightarrow B$ and is context switched out. Thread 2 completes pop $A \rightarrow B$. Later C is pushed and then A is pushed. Thread 1 resumes executing and the head of the list is 'still' A, so it completes its operation, changing the head of the list to B. Element C is now not linked as part of the list, without being popped. While mitigations can exist for recovering these elements, prevention is truly necessary.

To detect and prevent ABA sequences, an additional field is integrated into the pointers that can capture this sequencing information. This sequence number field carries the specific ordering of each operation being performed on the data structure. Thus, when a push or pop is attempted, the code reads the current pointer and extracts the sequence number. The next operation receives the subsequent sequence number. Then when the code attempts to update the data structure using compare-and-swap, not only must the pointer match but the sequence number must also match. Given that sequence numbers are finite, there is still the possibility that the sequence number has wrapped around at the same time that the original pointer is in place; however, with a sufficiently sized field, the chance of this occurrence is considered negligible.

Two follow up points need to be reviewed. Any physical system has a number of purely physical sources of error, so software errors below some threshold can be ignored. Second, should the system utilize load-linked / store-conditional instructions instead, it is not susceptible to the ABA problem, as these instructions depend on the memory location being accessed / modified and not detecting this through the value being changed.

11 TRANSACTIONAL MEMORY

Our approach so far to synchronization has been specifying how the system should complete the necessary operations. Transactional memory lets us turn this around to instead specify what needs to be completed safely and leaves the how to the system. In marking a region of execution as transactional, the system tracks every memory operation, such that when the transaction completes, it is as if all of the operations execute between the operations from any other processor. That claim may be confusing, so consider a simple example of two processors repeatedly writing to locations in memory. You may reasonably expect that the writes from the processors interleave in time, P0 then P1 then P0 and so forth. Transactionally, we want the operations not to interleave but all be P0 and then all from P1 (or vise versa). If the operations can be so ordered, then they have mutual exclusion with those memory locations.

The default way to ensure this transactional ordering is to aborting transactions that are violating this requirement. It is also possible for transactions to partial abort, rolling back to the last non-violating operation. And other work has put forward rebasing operations that can shift a transaction's operations to avoid the violation.

The tracked memory operations are treated as being in two sets: the read set and the write set. Locations in the a transaction's read set are permitted to overlap with other reads to the same location. Any write however will cause the

transaction to be aborted. Locations in the write set can not overlap with any other memory operation (either read or write) to those locations. Locations can be promoted from the read set to the write set.

11.1 Hardware / Software

To provide transactional memory to the programmer, some component must track and maintain the necessary information about memory usage to enforce the transactional semantics. There are two fundamental approaches: hardware and software. Hardware is already designed to track and share addresses via the coherence protocol, so extensions to track accesses transactionally are straightforward. Software approaches require instrumentation or a managed runtime that can intercept every memory operation, since any operation can disrupt a transaction.

While hardware is faster, there is some advantage to software based solutions. Each framework has to identify and then track each transactional access. The tracking part requires additional storage space, which can be a constraint with hardware approaches. In fact, experimental results in Section 11.4.2 show some transactions aborting solely due to these resource constraints. Some researchers have therefore proposed hybrid approaches, which allow the hardware to call into a software component to manage those transactions where the resources have been exceeded.

11.2 Undo / Redo Logging

Whether by software or hardware, the transactional memory system has to track the updates that are being transactionally made to memory. The system does so via logging. And the log is necessary to ensure that all of the updates are either fully applied to memory or were rolled back and did not happen. There are two choices in logging, does the system track the updates and not apply them, or does it apply the updates, but save the old results.

If the system keeps a log of the updates to be applied (redo logging), then rolling back the changes and aborting the transaction is as simple as discarding the log. However, the changes have not been applied to memory so committing the transaction can be time consuming as every logged memory operation must now be applied.

In the contrary design, undo logging, the system applied any updates to memory, while somehow protecting this memory from being globally accessed (otherwise the operations have effectively committed), but has to keep a record of the old values in each location. Then on a commit, the system stops protecting the locations and they are committed. But on the abort, the system has to use the log to reverse every memory operation.

Neither system design has a substantial advantage over the other. Instead, the best system will depend on the circumstances of the workload.

11.3 Optimistic / Pessimistic

A separate problem in designing a transactional memory system is when to detect conflicts in the transaction's read and write sets. There are again two basic approaches to the timing of this detection. Optimistically, the system only searches for conflicts when attempting to commit a transaction.

A separate question is on a conflict between two transactions, which transaction should be aborted. As each abort requires re-executing operations, transactions that have more operations may be preferred on a conflict. Alternatively, a transaction could also briefly ignore the abort in an attempt to complete. However, ignoring aborts can lead to livelock, thus this time needs to be bounded in elapsed time and not just progress.

```
void atomic_add(int* loc, int val)
{
  int status = XBEGIN();
  if (status == SUCCESS)
  {
    *loc += val;
    XEND(); // commit the transaction
  }
  else
  {
    lock();
    *loc += val;
    unlock();
  }
}
```

Fig. 11. Buggy Transactional Memory Usage

11.4 Intel TSX

A programmer can delineate the transactional section with XBEGIN and XEND instructions. Within the transactional section, all reads and writes are added to a read- or a write-set accordingly. The granularity for tracking is a cache line. If another processor makes a read request to a line in the write-set or either request to a read-set, then the transaction aborts.

Transactions can be semi-nested. A transaction can only commit if the outer transaction is complete. Internally nested transactions do not commit on XEND. If any transaction in the nest aborts, then the entire transaction aborts. If $|XBEGIN|$ equals $|XEND|$, then the entire transaction commits and becomes globally visible. Transactions can be explicitly aborted by the XABORT instruction, which enables the code to abort early when it can determine that the transaction will or should fail.

As I understand it, TSX is being built on top of the existing cache coherence mechanisms. Each cache line gains an additional bit to indicate if it is part of a transaction. Each memory operation is treated normally between the processor and the coherence hierarchy with several caveats. If a dirty, transactional block is evicted, then the transaction fails. If a dirty, transactional block is demand downgraded from modified to shared or invalid, then the transaction fails. In this case, a new message would indicate that the request to forward the data fails and the request should be satisfied by memory.

If the transaction commits, then the transactional bits are cleared on each cache line. And the lines operate normally according to existing cache coherence mechanisms.

11.4.1 Practical Concerns. As there are a variety of reasons for abort, some transactions can never complete successfully, so in practice, transactions need abort handlers that can complete the requested operations without using transactional memory. This can give arise to concurrency bugs, as the abort path must use a different synchronization mechanism than transactional memory. Even for a simple add to a global, see Figure 11, this code cannot just be wrapped in transactional memory and a mutex, and instead has a bug.

Assume that a thread has aborted the transaction and is running the lock version, it acquires the lock, and reads the old value of `*loc`. A second thread now enters this routine and starts the transaction. Its add can complete without

```
if (status == SUCCESS)
{
  if (test_lock() == LOCKED) XABORT();
  *loc += val;
  XEND(); // commit the transaction
}
```

Fig. 12. Fix for Buggy Transactional Memory Design

detecting a conflict with the locked version, thus committing a revised value that is also not detected by the locked version.

Therefore, we extend this design with an additional state that ensures the expected exclusivity. A new state is introduced that indicates whether the lock is held, which could be implicit within the lock itself or require some other variable. The following code sequence shows the modification that tests for the lock being held and treats the lock as failing the transactional sequence.

11.4.2 Experimental Results.

12 PROBLEMS

(1) In Figure 2 for test and test-and-set locks, how many coherence requests will be made if there are currently N threads waiting for the lock, and a N+1 thread attempts to acquire?