CSG280: Parallel Computing
Memory Consistency Models: A Survey in Past and Present Research

Jenny Mankin

December 2007

1 Introduction

The transition from single processor to shared memory multi-processors (or shared memory multi-core processor) has presented two challenges for shared memory design and implementation: keeping caches coherent and maintaining memory consistency [2].

The cache coherency problem occurs when one processor tries to access a region of shared memory which has been modified in another processor's private cache, but not yet written back to main memory, resulting in a read to memory which is out of date. This problem is fairly intuitive, and most computer architects can agree on using one of two cache coherency protocols: snooping (for shared memory) and directory-based (for distributed memory).

Memory consistency can be summarized as the properties which must be enforced among reads and writes to different locations in shared memory by different processors [10]. Likewise, a *Memory Consistency Model* is needed for a shared memory system to define the ordering in which reads and writes must be performed, both relative to each other in the same program and to other reads and writes by another program on another processor. The programmer must know the memory consistency model used by the hardware in order to ensure program correctness.

Though there are many solutions to the problem, each memory consistency model has its own advantages and disadvantages and it is not always clear which protocol works the best for a particular system and application. In this paper, I will discuss the problem of memory consistency, as well as several proposed solutions. The solutions come from research papers spanning the last 30 years, as some of the earliest models are still relevant today.

1.1 Memory Consistency Models

In order to maintain memory consistency in a multiprocessor system, a *Memory Consistency Model* is required to specify the memory behavior, or to provide a contract between the memory implementation and the program utilizing memory [15].

The reason memory consistency can be non-intuitive is that it violates the basic assumptions about memory in a sequential computer. For a serial program, the programmer can always assume that a *read* to a variable will return the value that was *last* written to its memory location, and a *write* to a variable will bind a value to its memory location; this value will be returned on all *subsequent* reads until the *next* write to that variable [6]. The definitions of "last", "subsequent", and "next" are constrained by the program order itself; compilers may be able to reorder instructions for optimization purposes, but it will only do so if data dependencies are maintained. In this way, a programmer can write a program knowing that the order in which code appears sequentially to them will be maintained through execution.

Unfortunately, this model of a memory system no longer holds true for multiprocessor systems. Definitions of words like "last", "next", and "subsequent" become unclear as multiple processors are writing to the same memory location in parallel. The following example, from [1] demonstrates how in the absence of an explicit program order, the program may not execute the way the programmer intends:

Initially:

```
data = 1000;
head = 0;
```

Processor 1:

```
data = 2000;
head = 1;
```

Processor 2:

```
while(head == 0) {
   ;    // Wait
}
varX = data;
```

Figure 1: Example code demonstrating how a program could run incorrectly if a memory consistency model is not provided or followed.

In this example, Processor 1 writes to two variables: first `data`, and then `head`. Processor 2 busy-waits until `head` is 1, then reads `data`. If program order were guaranteed, then it would be assumed that if `head` returns its most recent update (`1`), and because `data` was written to before `head`, the read to `data` would automatically return its most recent value (`2000`). Section 2 presents an example on how this guarantee would be violated, resulting in an inconsistent memory and incorrect program execution. In order to keep memory consistent, the programmer must know how consistent the processor promises to keep memory. If the memory is not maintained completely consistent, then the programmer must explicitly account for this. This is the point of the *Memory Consistency Model*: by providing the contract between hardware and software, the memory remains consistent and the program is guaranteed to execute correctly.

Memory consistency models can be broken up into several categories; in this paper, we explore three: The first is *Sequential Consistency*, in which the uni-processor model is extended to a multiprocessor system, with hardware forcing all memory operations to appear to execute one at a time.

The second category is *Relaxed Consistency* models, in which memory consistency is not maintained by hardware, but is enforced by the programmer with explicit synchronization mechanisms. Finally, consistency may be maintained through *Transactional Memory* models, in which the program is divided into logical "transactions", where transactions can perform memory accesses in parallel, but those memory accesses will not become permanent if a conflict occurs between transactions.

The rest of this paper will be broken up as follows: the three categories of consistency models will be discussed in Sections 2, 3, and 4, respectively. Section 5 will present a comparison of the models, with the advantages and disadvantages of each. As was suggested earlier, there is no clear-cut best model, so this survey paper will help the systems architect or parallel programmer understand what model might be best for their system or application. Section 6 will present my conclusions.

2 The Sequential Memory Consistency Model

From the programmer's point of view, the *Sequential Consistency Model* is the simplest to understand because it extends the uniprocessor model, and therefore, follows the basic assumptions which are made about sequential memory. The memory consistency is maintained through hardware, and therefore allows the programmer to write code which follows the intuitive memory model. The obvious benefit of this model is the ease of creating correct programs; however, this ease-of-use comes at a performance cost. Many common hardware and compiler optimizations can violate sequential consistency. As such, they must be eliminated, or additional steps must be taken to ensure sequential consistency.

Though most modern computing systems have trailed away from the sequential consistency model due to its inherent performance penalty, understanding of the sequential model will lead to a greater understanding of other models, as Relaxed Consistency Models are simply models from which the constraints of sequential consistency have been *relaxed*. For this reason, a detailed analysis of sequential memory consistency is presented here, along with examples and explanations of violations as provided by research and technical literature.

There are two requirements for maintaining sequential consistency [1]:

1. Program Order Requirement: Program order must be maintained among memory operations in a single processor.
2. Write Atomicity Requirement: A single sequential order must be maintained among all operations.

2.1 Program Order

The first requirement is fairly self-explanatory and intuitive, as it follows the same model as a sequential computer. Leslie Lamport, who is frequently credited [1, 15, 8] for first defining sequential consistency, presented an implementation-specific solution [13] to this first requirement; he suggested that each processor issues memory requests in the order specified in its program. When Lamport wrote his paper, Moore's Law was very much in effect, with the performance of computers doubling along with the number of transistors. Since that time, computer architects have been "cheating" Moore's Law with more advanced architectural features to improve performance further then simple increasing the clock frequency would allow. Thus, while a good suggestion given the technology at that point, this requirement is not enough today to guarantee sequential consistency. The following examples from [1] will show how some of these hardware optimizations will violate sequential consistency even though they adhere to Lamport's requirement, issuing memory requests in the order specified by the program.

There are three types memory operation pairs: a read-after-write, a write-after-write, and a read/write-after-read. Figure 2 provides an example of the first case: a read-after-write.

Initially:

```
    flagA = flagB = 0;
```

Processor 1:

```
    flagA = 1;
    if(flagB == 0)
        // Enter Critical Section
        flagA = 0;
```

Processor 2:

```
    flagB = 1;
    if(flagA == 0)
        // Enter Critical Section
        flagB = 0;
```

Figure 2: Dekker's Algorithm for Critical Sections: This code is guaranteed to execute correctly on a sequentially consistent system due to the *Program Order* requirement.

This algorithm is called *Dekker's Algorithm*; it protects the critical section without the use of mechanisms like locks, message passing, or mutexes. Each processor has its own flag. To enter the critical section, a processor must set its own flag and ensure that the other flag is not set. This algorithm is guaranteed to run on a sequentially consistent system. The reason for this is that, if there has been a write issued to the flag, that latest value is guaranteed to be stored in memory before the next read to it. Thus, both processors cannot simultaneously enter the critical section because both flags could not simultaneously be 0. Say, for example, that a write buffer (a common hardware optimization) is implemented on the processors, and the code is run in parallel. Now, both flags are set to 1, but with the writes delayed, both processes enter the critical section simultaneously. Therefore, a system with this optimization would not be considered sequentially consistent.

The example presented in the Introduction, Figure 1 presented an example which was, again, guaranteed to work on a system which maintained sequential consistency. With Processor 1 performing writes to `data` and `head` (the write-after-write case), and Processor 2 performing reads from `data` and `head` (the read-after-read case), this example demonstrates the other two memory pairs which must be maintained for the program order requirement of sequential consistency. Imagine the hardware is optimized with multiple memory modules to facilitate parallel memory operations, connected by a single general interconnect, and that the variables `head` and `data` are stored in separate modules. Different processors could have different access latencies reading from the separate memory modules. In this case, the system cannot guarantee that, for example, the read to `data` by Processor 2 will result in the most recent value, even if the read to `head` does. Once again, the adherence to a sequential consistency model would forbid the addition of performance-enhancing hardware optimizations. This disallowance of hardware optimizations is one reason why the use of a sequential consistency model declined in multiprocessor systems.

Another reason for the fading of sequentially consistent computers is that they are less able to utilize compiler optimizations—optimizations which, on any other system, would result in a significant performance improvement. Because program order must be maintained, the compiler cannot perform such common operations as instruction reordering, loop unrolling, and instruction elimination [1]. Another common compiler optimization—loading variables into registers—would cause program incorrectness because another processor could write to that memory location but the first processor would continue using the registered value. Both the elimination of useless statements and the register allocation problem can be solved with the `volatile` keyword, which indicates to the compiler that the value can change at any time by an external process or thread.

2.2 Write Atomicity

The second requirement for sequential consistency—write atomicity—forces any memory operation to appear to execute atomically to other memory operations. This requirement has two implications: writes to the same location happen in the same order to all processors, and a change to a memory location is observable to all processes simultaneously. A sequentially consistent system can be thought of as having a single memory connected to all processors by a single switch which connects a single processor to the memory at a time, with the switch providing global serialization among all memory operations [1]. This analogy could be traded for the implementation provided by Lamport, in his second requirement to create a sequential computer, when he stated that memory requests from all processors issued to a single memory module are services by one FIFO [13]. Again, however, Lamport's requirement does not reflect modern architecture. In this case, it is the addition of caches which cause problems which can violate memory consistency.

Adve (et al) declare that propagating a new value in memory to multiple caches is inherently a non-atomic operation, and must, therefore, be simulated by satisfying two conditions. The first condition is that writes to the same location be serialized, such that all processors see writes to the same location in the same order. If writes are seen in a different order on different processors, then sequential consistency will be violated. Observe the example presented in Figure 3, from [1].

In this example, both processors P1 and P2 write different variables to `varA`. Meanwhile, both processors P3 and P4 are waiting for the same conditions to be true—namely, that `varB` and `varC` equal 1—before writing the value of varA to private registers. If P3 and P4 see the writes to `varA` in a different order—with all other operations being equal—sequential consistency is violated since the writes to `varA` appear to be non-atomic. This situation could occur in any distributed shared memory system with an asymmetric topology resulting in differing delivery time for message depending on the origin and destination. A solution to this problem is to serialize all memory requests to the same location, as is done with Lambert's FIFO or Adve's suggestion of a single directory.

The second condition to ensure the appearance of write atomicity is to prohibit any reads from

Initially:

```
    varA = varB = varC = 0;
```

Processor 1:

```
    varA = 1;
    varB = 1;
```

Processor 2:

```
    varA = 2;
    varC = 1;
```

Processor 3:

```
    while(varB != 1) {;}  // Busy wait
    while(varC != 1) {;}  // Busy wait
    reg1 = varA;
```

Processor 3:

```
    while(varB != 1) {;}  // Busy wait
    while(varC != 1) {;}  // Busy wait
    reg2 = varA;
```

Figure 3: In a sequentially consistent system, this code is guaranteed to run correctly due to the *write atomicity* requirement.

occurring on any memory location for which there is an outstanding write; this can be accomplished with an acknowledgment of invalidates or updates sent by all processors. In a cache-based system, this condition ensures that values propagated to shared memory appear to reach all processors at the same time. This further explains the problem with the sequential consistency model: in order to utilize a cache, additional cycles must be spent waiting for an acknowledgment of a write, and performance suffers.

3 Relaxed Memory Consistency Models

Once it became clear that sequential consistency wasn't always necessary, and that the hardware overhead and performance degradation were not justified by ease-of-programming, there was a trend to create less strict models. The result is the group of memory consistency models known collectively as *Relaxed Consistency Models*, as these models *relax* one or more of the requirements of sequential consistency. Relaxed consistency models can be partitioned into subgroups using four comparisons: Type of Relaxation, Synchronizing vs. Non-Synchronizing, Issue vs. View-Based, and Relative Model Strength. These definitions will be used later in this section to classify examples of relaxed consistency models, wherever that definition is applicable. These four comparisons are described below:

- **Relaxation:** A simple and effective way of categorizing relaxed consistency models is by defining which requirement of sequential consistency is relaxed. Using the requirements for sequential consistency specified by [1], we can say that a relaxed consistency model either relaxes the program order or write atomicity requirement. When relaxing program order, the model may relax any or all of the orderings on memory operation pairs: read-after-write, write-after-write, or read/write-after-read. When relaxing write atomicity, a model may allow a processor to view its own writes before another processor can, or allow a processor to view another processor's write before the rest of the processors in the system can.

  For example, in Section 2.1, I discussed how a write buffer would violate sequential consistency for the example provided by Figure 2, Dekker's Algorithm. With this optimization, the processor could not guarantee a proper ordering in the case of a read-after-write. Therefore, we could say that the system provides a relaxed consistency model: The program order requirement will be relaxed in the case of a read-after-write. This relaxation is not a problem, once the programmer has knowledge of this consistency model and can account for it.

- **Synchronizing vs. Non-Synchronizing:** A synchronizing model divides shared memory accesses into at least two groups and assigns a different consistency restriction to each group. For example, one type of memory access may only need a weak consistency model, whereas another type of memory access may require a strict consistency model. A non-synchronizing model does not differentiate between individual memory accesses and assigns the same consistency model to all accesses collectively.

- **Issue vs. View-Based:** An issue-based relaxation focuses on how the ordering of an instruction issue will be seen by the entire system, as a collective unit. It is the belief of the authors of [15] that the purpose of an issue-based model is to provide an efficient interface for the programmer to simulate sequential consistency, trading some ease-of-use for performance. On the other hand, a view-based method does not aim to simulate sequential consistency; in this category of models, each processor is allowed it's own view of the ordering of events in the system, and these views do not need to match.

- **Relative Model Strength:** Some models are inherently more strong (or strict) than other models. Sequential consistency, described in Section 2, is one of the most strict, in that it has the least relaxations (none). Other models may be weaker, but compromise programmer effort. If rating the strength of a model by the relaxations of program order or atomicity, it may be possible to directly compare the strength of different models, as in a case where one model relaxes everything another model relaxes—plus more. Other times, models cannot be directly related, as they relax different requirements and a relative importance cannot be derived.

The rest of this section discusses specific relaxation models, and classifies them based on the above four sets of characteristics, if such a characteristic can be properly applied.

3.1 Weak Ordering

An early relaxed consistency model was presented in the mid-1980s by the authors of [4]. Their goal in this work was to categorize the problems created by highly-pipelined machines using memory prefetching or buffering. They point out that several high-end (for the time), general-purpose processors utilize pipelining of instructions to achieve parallelism. They may also implement prefetching of data and buffered writes to minimize the latency of a memory load or store, respectively. In the case of a buffered write, there may be a lock-up free cache, which does not block the processor on a miss. These are all examples of hardware optimizations which prohibited sequential consistency.

The next goal of the authors of this work was to create a simple programming model which would still allow the use of hardware optimizations to exploit instruction-level parallelism. For this, they introduce the concepts of *strong* and *weak* ordering. Strong ordering of memory accesses essentially implies that it maintains sequential consistency. The use for a weaker ordering, on the other hand, is based on the belief that sequential consistency does not need to be maintained within a critical section, as no other processors can access the memory within that critical section anyway until the critical section is exited. For this reason, synchronization is only necessary at certain points (aptly named "synchronization points"), thus providing the programmer with a simpler programming model without (all) the performance loss of sequential consistency. For the *Weak Ordering* consistency model, therefore, they distinguish all memory accesses as either "ordinary" or "synchronization" accesses. Then, according to that classification, ordinary accesses can be reordered amongst themselves by the hardware optimizations without loss of program correctness. Synchronizing accesses, on the other hand, are strongly ordered. Plus, access to a synchronizing variable is forbidden until all other shared data accesses have been performed, and access to global data is forbidden until a previous access to a synchronizing variable has been performed.

Though effort is spent on the programmer's part to distinguish all shared memory accesses, synchronization is simple once those accesses are labeled. The programmer needs only to protect access to the synchronization variables through the use of a critical sections. To the hardware, these synchronizing variables are differentiated only by the use of atomic instructions such as the *test-and-set* and *compare-and-swap* instructions.

As may be easily inferred by the language in the preceding paragraphs, this consistency model is a synchronizing, issue-based model. Within ordinary operations, it relaxes all three pairs of memory operations (read-after-write, write-after-write, and read/write-after-write); it also relaxes write atomicity by allowing a processor to read its own write before other processors can. It is clearly weaker than sequentially consistency.

3.2 Release Consistency

Release consistency, first discussed by the authors of [6], is a synchronizing model: all shared memory accesses are classified and labeled, and then associated with a particular consistency model based on their label. The idea of two types of accesses, one standard and one synchronizing, is borrowed from Weak Ordering. In this model, the authors further subdivide all operations. All shared memory accesses are either labeled *special* or *ordinary*; *special* operations may be competing (ie, there are two or more accesses to the same memory location and at least one of the accesses is a store) while *ordinary* accesses are not competing. Special operations are further subdivided as either *synchronizing* or *non-synchronizing*. As with [4], synchronizing accesses are performed within critical sections and are protected by locks. In this same way, the authors of Release Consistency subdivide synchronizing variables into either *acquire* or *release*, similar to the functionality of acquiring and releasing a lock.

Once all memory accesses are labeled according to the above classification, they can be reordered depending on the consistency model used for their class. There are, therefore, only three conditions for Release Consistency [6]:

1. All previously-occurring *acquire* accesses must be performed before any *ordinary* access is allowed to perform (with respect to any processor).
2. All previously-occurring *ordinary* accesses must be performed before any *release* access is allowed to perform (with respect to any processor).
3. *Special* accesses are merely processor consistent with each other.

Condition 3 for Release Consistency could have been that special accesses remain sequentially consistent with each other, but the authors found that *processor consistency* (discussed in the next subsection, it is weaker than sequential consistency) gave them the same correct results with improved performance. Condition 3 also implies that non-synchronizing accesses must retain the higher level of processor consistency.

Again, from the vocabulary above it can be surmised that this model, too, is a synchronized, issue-based model. Again, all three program order memory access pairs are relaxed, as are both requirements of write atomicity. It is weaker than both Sequential Consistency and Weak Ordering, as Release Consistency's three conditions ease the restrictions used for Weak Ordering.

3.3 Processor Consistency

Unlike the previous two models, Processor Consistency, first introduced in [8], is both view-based and non-synchronizing. In other words, each processor is allowed to have its own view of the system, and all memory accesses are treated the same. In his original paper, Goodman specifies that the order in which writes are observed must be the same as the order in which they were issued. However, if two processors issue writes, those writes do not need to appear to execute in the same order, from the perspective of either of the two processors or a third processor. The authors of [6] specify the conditions of Processor Consistency in another way; they state that:

1. Before a load operation is allowed to perform with respect to any other processor, all previous load accesses must have already been performed.
2. Before any store operation is allowed to perform with respect to any other processor, all previous loads and stores must have been performed.

The two conditions above imply one important fact: only the read-after-write program order requirement is relaxed. Therefore, while the example provided in Figure 2, Dekker's Algorithm, was guaranteed to work in a sequentially consistent system, it would not work in a Processor Consistent system. In order to guarantee correct operation, the programmer would have to use an additional synchronization mechanism, such as a *test-and-set* atomic lock or a fence/barrier operation. In his paper [8], Goodman states that many processors of the time that may have *appeared* to be Sequentially Consistent were *actually* only Processor Consistent.

To summarize, the Processor Consistency model is non-synchronizing and view-based. It relaxes only the read-after-write program order requirement, but relaxes both write atomicity requirements in allowing a processor to read both its own and other processors' writes early. It is weaker than Sequential Consistency and stronger than Release Consistency, but cannot be related to Weak Ordering.

4 Transactional Memory Models

After the development of many Relaxed Consistency models, researchers sought out a way to combine both cache coherency and memory consistency models in a single, hardware or software-supported communication model for shared memory that is easy for programmers to use. This is accomplished with Transactional Memory models. A *transaction* is a sequence of operations executed by a single thread. Once the operations have been completed, the transaction does one of two things: If there are no memory operations which conflict with memory operations of another transaction, the transaction *commits* and it takes effect; otherwise, it *aborts* and its effects are discarded. While the previously-discussed memory consistency models require a cache coherency model as well to guarantee correct operation of the whole system, a transactional memory model has both memory consistency and cache coherency built in. It also provides an easier-to-use model than relaxed consistency models, but does not suffer the performance penalty of a sequential consistency model (if done properly).

The system-wide arbitration to commit or abort a transaction can be done either through hardware, software, or some hybrid mechanism. While some of the first powerful transactional memory models were conceived using hardware, there are several intelligent software models which are especially useful when there is no hardware support but the developer still prefers to use a transactional memory model over any other memory consistency model. In this section, I will discuss examples of each implementation. Subsection 4.1 will focus on hardware implementations and Subsection 4.2 will focus on software implementations of transactional memory models.

4.1 Hardware Transactional Memory

4.1.1 Lock-Free Transactional Memory

In 1993, the authors of [12] presented the first use of a transactional memory model for a general purpose system. Their goal was to make lock-free synchronization as easy to use and efficient as a lock-based implementation. Since lock-based implementations are prone to priority inversion, convoying, and deadlock, this will result in an overall more-efficient system. This implementation differs from a typical database transaction in that transactions here should be short-lived and access only a limited amount of memory; in general, they state that a transaction should be able to complete in a single scheduling quantum, and the number of memory locations accessed should not be greater than the cache which is added to handle accesses for a single transaction.

This implementation requires the addition of several instructions to the instruction set (the hardware added to support these instructions is discussed next). There are three primitive instructions to access memory: LT reads one word of memory into a private register, LTX also reads one word of memory into a private register but also hints that the location will likely be updated, and ST writes a value from a private register, though the change does not become permanent until the transaction commits. The authors define the *read set* as all locations read by an LT instruction, and the *write set* as all locations read by LTX or written to with ST. These definitions are used for the next set of primitive instructions: those which can manipulate transaction state. There are three such instructions: COMMIT attempts to make the transactions changes permanent, succeeding only if no other transaction simultaneously updated any location in the read or write set and no other transaction read a value contained in this transaction's write set. The ABORT instruction discards all updates to the write set. Finally, the VALIDATE instruction (added only to ensure that a transaction which will eventually abort does not perform illegal operations such as divide by zero, access out-of-range memory, etc) merely tests whether the current transaction would have reason to believe it would abort if it were committed, and if so, effectively aborts.

The architecture required to support the added instructions is simple: it only requires an extra cache and some additions the existing cache coherence protocol. Non-transactional operations use the same hardware and protocols they would in the absence of a transactional memory implementation; no additional hardware is used. For transactional operations, a *transactional cache* is required; this cache is restricted to the primary level (L1) of the cache hierarchy. This cache holds all tentative writes without propagating them to other processors or main memory until a commit is successful. At that point, the cache lines may be snooped by other processors or written back to main memory after a cache replacement. The implication of this is that committing and aborting transactions are local to the cache; this results in a faster system since all cache lines do not need to be written back at transaction commit, resulting in a memory bottleneck. The *transactional cache* has all states associated with caches in Goodman's snoopy protocol [7] (INVALID, VALID, DIRTY, and RESERVED), but it also has additional states (EMPTY, NORMAL, XCOMMIT, XABORT) used to maintain the status of the accesses in a transaction. They modify the cache coherence protocol by adding additional bus cycles (T_READ, T_RFO, and BUSY) to the standard bus cycles (READ, RFO, WRITE) of Goodman's original protocol. Finally, each processor maintains a TACTIVE and TSTATUS flag for each transaction. Because the ability to commit and abort transactions is critical to performance, the access to both regular and transactional caches must be performed in a single cycle, including commits and aborts.

The authors compare their transactional memory implementation to several other common hardware and software mechanisms used for synchronization. These mechanisms are listed in Table 1. The authors run three benchmarks on each synchronization mechanisms; in two of the benchmarks, their implementation has the best performance, and in the third, is only beaten by the LL/SC mechanism (though they state that the advantage LL/SC gained will be lost on any transactional object which spans more than one word). With these results, the authors show that a lock-free hardware transactional memory scheme can achieve performance equal to or greater than that of other implementations, but with an easier programming model and guarantee of program correctness.

4.1.2 Transactional Memory Coherence and Consistency (TCC)

Another hardware transactional memory implementation is provided in [9]; this research presents a well-regarded hardware implementation of Transactional memory Coherence and Consistency (TCC). In this work, the authors borrow from the database management concept of *optimistic concurrency*, modifying it for coarser granularity to make the model more effective for general purpose programming. Their model follows many of the semantics of the previously-discussed transactional model work of [12], though they replace the concept of using transactions only at critical sections with using them at all times. They also draw from the work on thread-level speculation, utilizing

Synchronization Mechanism	*Description*
Spin Locks (software)	Each processor repeatedly tries a test-and-set operation until it acquires the lock. Utilizes exponential backoff after unsuccessful lock acquisition.
Test-and-test-and-set (software)	Each processor repeatedly reads the cached value of the lock until observing the lock is free, then directly applies test-and-set to the lock in memory. Requires cache-coherent protocol. Utilizes exponential backoff after unsuccessful lock acquisition.
Software Queueing (software)	A process previously unable to acquire a lock itself places itself on a software queue, and thus no longer has to poll the lock.
Load_Linked / Store_Cond (LL/SC) (hardware)	LL operation copies the value of a shared value to a local variable. SC changes its value in memory only if no other process had modified that value in the interim.
Hardware Queueing (hardware)	Like software queueing, except that the queue maintenance is incorporated into the cache coherence protocol itself.

Table 1: Description of hardware and software synchronization mechanisms tested against the original Hardware Transactional Memory implementation.

---

many of the same concepts and hardware for their transactional memory implementation.

In this work, memory consistency is maintained by providing a sequential ordering between transaction commits, as opposed to the more fine-grained previous idea of maintaining order between all individual memory operations. The authors define all memory operations that happened in a first committing transaction as happening "before" all memory operations that happen in a second committing transaction, regardless of the ordering they actually took. They also specify that if a processor reads data which later is updated by another processor's commit, the first processor must violate and roll back. Therefore, with a lot of interaction between transactions, performance could greatly suffer. Coherency is also only maintained at transaction commits. While transactions are running simultaneously, they can legally hold the same cache line, modified, in different caches. At commit, however, a broadcast notifies all other processors of cache changes that occurred in that transaction, at which point each processor can either invalidate or update the corresponding cache lines.

The advantage of hardware transactional memory is the ease-of-effort on the programmer's part. The TCC parallel programming model, in its most basic form, consists of only three steps:

1. **Divide into Transactions:** In this step, the programmer coarsely divides the program into transactions. There is only one explicit rule: The programmer must never insert a transaction break between a load and a subsequent store of a shared value. Other than this rule, the programmer does not need to guarantee the independence of transactions, as the hardware will take care of this at runtime.

2. **Specify Order:** In this optional step, the programmer specifies an ordering between transactions by assigning hardware-managed *phase numbers* to the transactions; transactions may then only commit in the order of their phase numbers. In the absence of this step, transactions will commit in whatever order they complete their operations.

3. **Performance Tuning:** In this optional step, the TCC system provides feedback on violations between transactions. The programmer can then follow some guidelines to optimize performance by changing transactions' sizes.

The TCC system requires basic speculative execution hardware (such as checkpointing hardware and memory monitoring bits) to manage speculative buffering of memory references and commit arbitration. The authors also provide extensions and improvements to TCC to improve performance and add extra functionality. To improve performance, they present double buffering to allow a processor to continue execution instead of stalling when a transaction is committing or waiting to commit, hardware-controlled transactions to mark transaction boundaries or order transactions, and presenting hints to the hardware when a memory operation is local to reduce the number of broadcasts. For extra functionality, the authors present a mechanism to allow I/O, as it would destroy program correctness if a transaction read data from I/O, then was aborted, losing the data read from I/O.

While not showing a direct comparison with relaxed consistency models, the authors show that their implementation results in good performance for specific system requirements. However, they find that the limiting factor in their implementation is the broadcast bandwidth among all nodes. They conclude, therefore, that this implementation would only work on a CMP or a single-board multiprocessor, but could not scale well due to the system-wide arbitration necessary for commits and the broadcast bandwidth. Compared to the previous implementation of HTM, this model is clearly easier to use as the transactions are defined in a coarser granularity. A naive implementation might result in a decreased performance, however, if there are many transaction aborts. Fortunately, this risk is mitigated by the additional features and guidelines provided by the authors to reduce transaction conflicts.

4.2 Software Transactional Memory

4.2.1 Dynamic Software Transactional Memory (DSTM)

A decade after first presenting hardware transactional memory as a means of providing memory consistency and coherence [12], Herlihy developed a Software Transactional Memory (STM) model, the first to allow transactions and transactional objects to be created at runtime [11]. This paper concentrates on a Java ^{TM}implementation; they also have a C++ implementation but do not discuss it due to additional complexity from memory management. They argue that their implementation is simplified by using a non-blocking progress condition called *obstruction freedom*; this condition guarantees progress only in the absence of contention, and therefore, does not rule out livelock. As an aside, the author of [5] argues for an even simpler model, showing that obstruction freedom and non-blocking transactions are unnecessary for shared memory systems, and abandoning it will result in a faster STM implementation.

The Dynamic Software Transactional Memory model (DSTM) presented in this paper works by creating *transactional objects* which can access transactions. Transactional objects (called TMObjects) are simply containers for regular Java objects, and can be thought of as a *version* of the transaction. Each TMObject is a pointer to a *locator* object; the locator object has three fields: *transaction* points to the most recent transaction to open the TMObject (each transaction will have a status of either WRITE, ACTIVE, or COMMITTED), *new object* points to the new object version, and *old object* points to an old object version. This structure can be seen in Figure 4. Out of the *old* and *new* versions, the object which is currently valid depends on the status of the transaction which most recently opened the TMObject in WRITE mode. If that transaction is:

- **Committed**: The *new* object is the valid object and the *old* object is meaningless.
- **Aborted**: The *old* object is the valid object and the *new* object is meaningless.
- **Active**: The *old* object is the current version and the *new* object is the tentative version which may be committed or aborted.

The reason the TMObject is a pointer to a locator pointing to the three fields (put another way, the reason there are two levels of indirection) is because the authors need a way to atomically change all three fields when a new transaction object is opened. By having the TMObject point to a locator object, they can just do an atomic compare-and-swap (CAS) to swing the start pointer from one locator object to another, effectively changing all three fields atomically. This process is shown in Figure 4. In [3], the authors claim that this second level of indirection is costly, and that a more efficient implementation would utilize only a single level of indirection.

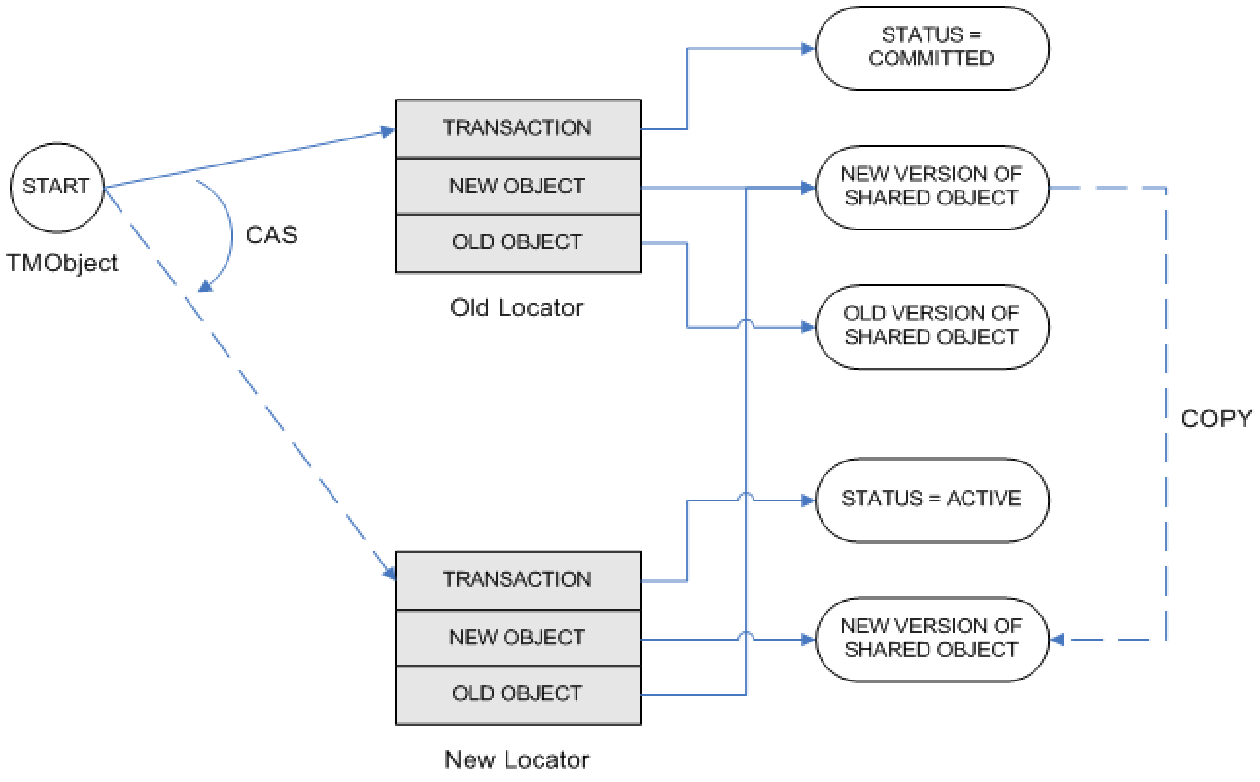

Figure 4: Opening a new transactional object (TMObject) in DSTM a after previous transaction commit.

DSTM presents several interesting features to reduce contention among transactions. First, a transaction can determine if it is about to abort another transaction, and a contention manager can determine whether to allow it continue or wait to allow the other transaction to complete. The contention manager can exploit specific information about time, operating systems services, scheduling, and hardware. DSTM also allows the user to open transactions in read-only mode; multiple transactions can then open the same transactional object for reading without accounting for conflicts. Finally, they allow for the early (before commit) release of an object which has been opened in READ mode. With this mechanism, other transactions accessing that object will no longer conflict with the releasing transaction when they reach their own commit. Though this method is powerful, it is also dangerous as its incorrect use could result in an inconsistent system.

This paper shows a very powerful way of creating a transactional memory model entirely in software. Some of ideas contained within it may not be optimal, such as the double levels of indirection, the use of a slow and costly implementation in Java, and the decision to make the model obstruction free. However, it is a solid model and provides researchers a starting point for improvement. In fact, many models created after this paper credit these authors for their fine work even as it is improved upon.

4.2.2 Transactional Locking (TL)

The authors of [3] performed a detailed analysis of existing software transactional models (STMs) and used the results to develop an implementation which they deem to be superior to the others. They call their implementation "Transaction Locking" (TL) and claim that it is the simplest and best performing "to date" (2007). They focus on small transactions, stating that large transactions may be too optimistic, and could create additional trouble when the transactions embed I/O operations which cannot be rolled back at abort time.

In their initial tests, these researchers made several observations on what makes software transactions fast. First, they found that non-blocking transactions were less efficient than lock-based transactions in several instances. Because of this, they utilize hardware locks, and thus consider their software transactional model more of a hybrid model (HyTM) (though other authors also use locks and still consider their models to be STMs). They find that implementations in which locks are acquired as transactions encounter them (they call these *encounter order* transactions) perform well on data structures with little contention but do not perform well with high contention. For this reason, they implement both *encounter order* and *commit-time* transactions—which only acquire locks when the transaction is committing. While other implementations allow mechanisms for transactions to abort other transactions to ensure progress, these authors claim these mechanisms are unnecessary and add significant cost; they use a simple time-out to abort threads which may be taking too long.

With every memory location contained in a transaction, there is a corresponding versioned lock; the version increments at every successful lock release. Locks are acquired with an atomic compare-and-swap operation. Locks are acquired at various granularities, though the authors prefer using one lock per shared object. In both encounter-order and commit-mode implementations, the TL algorithm maintains a linked-list of thread-local read and write sets. A read-set entry is defined as containing the address of the lock and version number of the acquired lock for the transactionally loaded variable. The write-set contains the address of the variable to be transactionally written, the value to be written, and the address of the variable's lock. The following list summarizes the stops taken to execute a transaction using the author's preferred commit-mode algorithm:

1. Run the transactional code; acquire locks for shared locations, and build read and write sets. Before a transactional load, use a Bloom filter to see if the load address is already in the write set. If so, return the last value written. If not, acquire the lock associated with the variable, save the version to the read set, and fetch the actual variable. Periodically during the transaction, validate the read-set and abort if invalid.

2. Attempt to commit the transaction. Acquire locks of all locations to be written. If the lock also appears in the read set, then atomically acquire the lock *and* validate that the version in the read-entry matches the most current version; if there is a mismatched version, abort and retry the transaction. To avoid deadlock, abort the transaction and retry if a a timeout occurs before all locks are acquired.

3. Ensure version numbers haven't changed in all read-only memory locations. If a version number has changed, release the lock, abort, and retry the transaction.

4. This step will only be reached once the transaction has committed. Write-back all entries from the local write set to the proper locations in shared memory.

5. Atomically release all locks in write set and increment the version number.

The authors compare four variations (encounter-order vs. commit-mode, and per-object vs. per-stripe lock granularity) on their transactional-locking implementation with many other past implementations of memory consistency and transactional memory models. The comparisons include: mutual exclusion locks (mutex, spinlock, etc), a non-blocking STM, a lock-based/encounter-order STM, a Compare-and-Swap lock-free skiplist, and a hand-crafted lock-based concurrent red-black tree (also called "Hanke"). In their first benchmark, the CAS-based implementation has twice the throughput of any other implementation, with the authors' own encounter-order/per-object locks performing the best on large data structures. In the next set of tests, "Hanke" performs well for low-contention but degrades with more contention. The authors' TL with commit mode/per object locking performs well the entire time and scales well for high contending large data structures and increasing number of threads.

While the authors cannot always beat the performance of a hand-crafted implementation like Hanke, they can provide a strong-performing implementation which provides programmer's ease-of-use and is generic enough to perform well across a range of benchmarks, contention, and inherent parallelism. The authors' detailed comparison between many older implementations of STM is valuable both in itself and in the way they use use results to develop a strong algorithm of their own.

5 Memory Consistency Model Comparisons

In this paper, I presented a summary of some past research into memory consistency models. Within the three categories of models (Sequential, Relaxed, and Transactional Memory), I was able to discuss the relative strengths and weaknesses of certain models. Across the different categories of models, the comparison is a little more difficult. There is no one best model, only the model that best fits the application, architecture, and programmer.

The sequential memory model is the easiest model to use for parallel computing. For an novice programmer who does not fully understand the problems inherent with memory consistency and cannot implement explicit mechanisms for synchronization, a sequentially consistent system may be the only possible model that will result in program correctness. There will be a significant penalty in performance, however, which could negate even the advantage of using a parallel system in the first place. Across a wide range of applications, the Relaxed Consistency models would average the best performance, but at greater programmer effort. A programmer with extensive knowledge of the architecture and experience in parallel programming will be able to write a high-performing application while ensuring correctness with synchronization mechanisms like barriers, locks, and fence instructions. Transactional memory, on the other hand, may perform very well in certain situations and less well in others, depending on program and system requirements. For example, a Transactional Memory model may not be well suited for a real-time system in which which transaction conflict and subsequent abortion would cause a process to miss its hard real-time deadline. Additionally, choosing the right transactional model would be critical in obtaining that strong performance. For example, TL could handle high contention among transactions when encounter-order locking algorithms could not, and TCC could handle I/O but TL could not. Additionally, while HTMs ultimately provide the best performance, without the added hardware the models are useless; in such cases a STM is the only option. Given all of these factors, there is no clear-cut best memory consistency model: only a best model for a given application, system, and programmer.

6 Conclusions

In this paper I discussed Memory Consistency and Memory Consistency models. I discussed the necessary requirements for the simple-but-low-performance Sequential Consistency Model and its requirements, and discussed how hardware and compiler optimizations prevent it from performing well. I then discussed Relaxed Consistency models as a class, and how they can be divided into categories based on what sequential consistency requirements they relax, whether they are synchronizing or non-synchronizing, whether they are issue-based or view-based, and their relative strengths. I discussed three examples of Relaxed Consistency models: Weak Ordering, Release Consistency, and Processor Consistency. Next, I transitioned to a more modern topic, transactional memory, and discussed the basic characteristics of a transactional memory model before detailing specific implementations of hardware and software transactional memory models. Among the three categories, each model has its own strengths and weaknesses; for this reason, there is no clear cut "best" model for all architectures and applications. For a given system, application, and programming team, the only way to find the best model is to determine which models can and cannot be used, and to weigh the relative advantages and disadvantages of those which can to find the best solution.

References

[1] Sarita V. Adve and Kourosh Gharachorloo. Shared memory consistency models: A tutorial. *IEEE Computer*, 29(12):66–76, 1996.

[2] David E. Culler and Jaswinder Pal Singh. *Parallel Computer Architecture: A Hardware/Software Approach.* Morgan Kaufmann Publishers, Inc., 1999.

[3] Dave Dice and Nir Shavit. Understanding tradeoffs in software transactional memory. In *CGO '07: Proceedings of the International Symposium on Code Generation and Optimization*, pages 21–33. IEEE Computer Society, 2007.

[4] Michel Dubois, Christoph Scheurich, and Faye Briggs. Memory access buffering in multiprocessors. In *ISCA '86: Proceedings of 13th Annual Symposium on Computer Architecture*, pages 434–442. IEEE Computer Society Press, June 1986.

[5] Robert Ennals. Software transactional memory should not be obstruction-free. Technical Report IRC-TR-06-052, Intel Research Cambridge, January 2006.

[6] Kourosh Gharachorloo, Daniel Lenoski, James Laudon, Phillip Gibbons, Anoop Gupta, and John Hennessy. Memory consistency and event ordering in scalable shared-memory multiprocessors. In *ISCA '98: 25 Years of the International Symposia on Computer Architecture (Selected Papers)*, pages 376–387. ACM, 1998.

[7] James R. Goodman. Using cache memory to reduce processor-memory traffic. In *ISCA '83: Proceedings of the 3rd International Symposium on Computer Architecture*, pages 124–131. IEEE, June 1983.

[8] James R. Goodman. Cache consistency and sequential consistency. Technical Report 61, IEEE Scalable Coherent Interface (SCI) Working Group, March 1989.

[9] Lance Hammond, Vicky Wong, Mike Chen, Brian D. Carlstrom, John D. Davis, Ben Hertzberg, Manohar K. Prabhu, Honggo Wijaya, Christos Kozyrakis, and Kunle Olukotun. Transactional memory coherence and consistency. In *ISCA '04: Proceedings of the 31st Annual Internation Symposium on Computer Architecture*. IEEE Computer Society, 2004.

[10] John L. Hennessy and David A. Patterson. *Computer Architecture: A Quantitative Approach*. Morgan Kaufmann Publishers, Inc., 3 edition, 2003.

[11] Maurice Herlihy, Victor Luchangco, Mark Moir, and William N. Scherer III. Software transactional memory for dynamic-sized data structures. In *PODC '03: Proceedings of the 22nd Annual Symposium on Principles of Distributed Computing*, pages 92–101. ACM, July 2003.

[12] Maurice Herlihy and J. Elliot B. Moss. Transactional memory: Architectural support for lock-free data structures. In *ISCA '93: Proceedings of the 20th Annual International Symposium on Computer Architecture*, pages 289–300. ACM, 1993.

[13] Leslie Lamport. How to make a multiprocessor computer that correctly executes multiprocessor programs. *IEEE Transactions on Computers*, (9):690–691, September 1979.

[14] Nir Shavit and Dan Touitou. Software transactional memory. In *PODC '95: Proceedings of the 14th Symposium on the Principles of Distributed Computing*. ACM Press, 1995.

[15] Robert C. Stienke and Gary J. Nutt. A unified theory of shared memory consistency. *Journal of the ACM*, 51(5):800–849, 2004.