Modeling Weakly Consistent Memories with Locks

Victor Luchangco
Massachusetts Institute of Technology
victor_l@theory.lcs.mit.edu

Data races, or unsynchronized access to shared data, produce behavior that is difficult to reason about. A common technique for eliminating data races is to designate regions of code as critical, and to use locks to ensure that at most one thread executes critical code at any time. On systems that allow processors to see inconsistent views of memory, special care must be taken to synchronize operations that implement locks. Otherwise, the locks may not guarantee mutual exclusion, or even if they do, a thread that enters its critical section may not see the effects of a thread that already exited its critical section.

To reason about these systems, weak consistency models have been proposed [1, 3]. Unfortunately, experience has shown that it is difficult to understand and use these models, leading some researchers to advocate the simpler, stronger models [6]. One problem with the proposed models is that they are *processor-centric*; that is, each processor requests a sequence of operations, and there are no control dependencies across processors. Another problem is that they model only low-level synchronization, and do not take advantage of the greater structure provided by locks. These models are appropriate for specifying hardware guarantees, but do not capture the richer structure available in parallel languages such as Cilk [11] or Java [5].

We show how to model weakly consistent shared memory systems with locks in a *computation-centric framework* [2], in which the interface between the program and memory is expressed by a *computation*, specified by the program, and the values returned for the operations of the computation. A computation captures the programmer's intention—whether, for example, an operation logically depends on a prior operation or merely occurs later, or whether a memory operation acquires or releases a lock or is an "ordinary" memory access—rather than details dependent on timing and other unpredictable variables. We find this framework makes it easy to specify and reason about the consistency guarantees of a system.

Within this framework, we define *weak sequential locking*, a new consistency model for a system with locks. Weak sequential locking has very weak guarantees. In the absence of locking, only operations that are explicitly ordered by the program must be seen in that order by all threads. Locks guarantee mutual exclusion, but lock accesses synchronize on a *per-lock basis only*. This is in contrast to release consistency or weak ordering [3, 1], in which synchronization is global. Despite the weakness of these guarantees, a data-race-free program executes on a system with weak sequential locking just as on a system that guarantees sequential consistency.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
SPAA 2001 Crete, Greece
Copyright 2001 ACM 0-89791-88-6/97/05 ...$5.00.

Computations

A ***computation*** C with locks L consists of a dag (V_C, E_C) and a ***locking function*** $\Lambda_C\colon V_C \times L \to \{\text{ACQ}, \text{REL}, \text{NIL}\}$. The vertices V_C of the dag are operations; the edges E_C encode ***precedence dependencies***, which indicate that one operation must be applied before another. The transitive closure of the precedence dependencies is the ***program order***, denoted by $\prec_C$. The program order defines a logical notion of time: An operation x ***precedes*** another operation y in a computation C if $x \prec_C y$. Two distinct operations are ***concurrent*** in C if neither precedes the other.

The locking function indicates which locks each operation acquires or releases. An operation x ***acquires*** $l \in L$ in C if $\Lambda_C(x, l) = \text{ACQ}$; it ***releases*** l if $\Lambda_C(x, l) = \text{REL}$. The set of operations that acquire l is $Acqs_C(l)$, and the set of operations that release l is $Rels_C(l)$. An operation that acquires a lock, and any operation that follows it in the program order holds the lock until it is released; that is, y ***holds*** $l \in L$ ***acquired by*** $x \in Acqs(l)$ in C if $x \preceq_C y$ and there is no $z \in Rels(l)$ such that $x \prec_C z \prec_C y$. For $x \in Acqs(l)$, the set of operations that hold l acquired by x is a ***locked section*** for l, or an ***l-section***, denoted by $Holds_C(l, x)$. We say that y ***holds*** l if it is in some l-section. We denote the set of operations that hold l by $Holds_C(l)$.

Operations that acquire and release locks come in matching pairs; that is, for each $l \in L$, there is a bijection $r\colon Acqs(l) \to Rels(l)$ such that $Holds(l, x) = \{y : x \preceq_C y \preceq_C r(x)\}$ for all $x \in Acqs(l)$. For simplicity, we also prohibit concurrency within a locked section; that is, $Holds(l, x)$ is totally ordered by $\preceq_C$. A computation is ***well-formed*** if these two conditions hold for all $l \in L$.

Two operations ***conflict*** if the order in which they are done may affect the result of the computation. For a read/write memory, this means that they access the same location, and at least one of them is a write. Concurrent operations that conflict comprise a ***race***. Races are sometimes necessary for communication between threads, in which case, the program synchronizes the operations, usually using locks.

A lock l ***arbitrates*** a race if both operations hold l. A ***data race*** is a race that is not arbitrated by any lock. A computation is ***data-race-free*** if it has no data races; that is, if all races are arbitrated by some lock. We denote the set of data-race-free computations by *DRF*. A program is ***data-race-free*** if it generates only data-race-free computations.

Memory models

A computation together with the return values for the operations comprise an observation of a program running on the system; that is, an ***observation*** of the memory is a pair (C, ρ), where C is a computation and $\rho\colon V_C \to \mathbb{R}$ assigns a return value to each operation. A ***computation-centric memory model*** for a memory system is a set of observations of the system. If M is the model for a memory system, then $(C, \rho) \in M$ is a possible observation of that system: when executing C, the system may return $\rho(x)$ for each operation $x \in V_C$.

A serial system determines return values for a computation by *scheduling* the operations, and executing them according to the schedule. A ***schedule*** is a total order on the operations that respects the constraints of the computation: It should be consistent with the program order, and each lock should be alternately acquired and then released. Formally, if α is a schedule and $\alpha|_l = x_1, x_2, \ldots, x_k$ is its projection onto the operations that acquire or release l, then $\Lambda(x_i, l) = \text{ACQ}$ if i is odd, and $\Lambda(x_i, l) = \text{REL}$ if i is even. We say that the schedule ***respects locking*** for C. We denote the set of schedules of C that respect locking by $RspLk(C)$.

For any schedule α, we use $retval(x, \alpha)$ to denote the value returned for x when α is applied to the memory. We say that a schedule α ***explains*** a return value ν for $x \in V_C$ if $retval(x, \alpha) = \nu$. A concurrent system may not have a single schedule that explains all the return values. If it does, we say the system is *sequentially consistent* [8]. Formally, the ***sequential consistency with locking*** model is

$$SCL = \{(C, \rho) : \exists \alpha \in RspLk(C), \forall x \in V_C, \rho(x) = retval(x, \alpha)\}.$$

When programming systems with locks, most programmers assume sequential consistency with locking.

Sequential consistency is expensive to implement because it requires all operations to synchronize with each other.[1] For data-race-free programs, sequential consistency can be maintained by synchronizing only those operations that implement locks. Because programmers avoid data races even in sequentially consistent systems, several weak consistency models have been proposed that take advantage of this observation [3, 1].

Unfortunately, these models describe the system at a low level, which is useful for reasoning about the hardware, but difficult for programmers [6]. We propose a new model, *weak sequential locking*, that consistently orders all acquires of the same lock. Formally, a ***per-lock total access order*** $\prec_L$ of C is a partial order on the operations V_C consistent with the program order such that $\prec_L$ totally orders $Acqs(l)$ for all $l \in L$. The set of per-lock total access orders of C is $PLO(C)$. An observation (C, ρ) satisfies weak sequential locking if there is a per-lock total access order $\prec_L$ such that for all $x \in V_C$, $\rho(x) = retval(x, \alpha)$ for some schedule $\alpha \in RspLk(C)$ that is consistent with $\prec_L$. We use $Cons(\alpha, \prec_L)$ to mean that α is consistent with $\prec_L$. Then the ***weak sequential locking*** model is

$$WSL = \left\{ (C, \rho) : \begin{array}{r} \exists \prec_L \in PLO(C), \forall x \in V_C, \exists \alpha \in RspLk(C), \\ Cons(\alpha, \prec_L) \wedge \rho(x) = retval(x, \alpha) \end{array} \right\}$$

Weak sequential locking is similar to *scope consistency* [7], in that memory updates are only propagated from threads that release a lock to threads that acquire the same lock. This reduces the amount of communication required compared with release consistency, which must propagate memory updates to any thread that acquires any lock. Although this may not be significant for hardware implementations of shared memory, it can be for software implementations. Scope consistency is still processor-centric, however, and so it cannot capture cross-processor dependencies easily.

Our main result is that a data-race-free program cannot distinguish weak sequential locking from sequential consistency with locks. We say that one memory model ***implements*** another under a set of computations if, for each computation in the set, every observation of the first with that computation is also a computation of the second. Our theorem then is:

THEOREM 1. *WSL implements SCL under DRF.*

This theorem says that a programmer can assume the memory guarantees sequential consistency with locking as long as it guarantees weak sequential locking, and the program is data-race-free. Because programs with data races are generally considered to be bugs, even when running on a sequentially consistent memory with locking, this should not be a significant burden on the programmer. Yet, it allows the memory to achieve significant gains in performance.

The results of this paper can be generalized to allow the memory to have an arbitrary data type, to incorporate computations with concurrency within locked sections, and to use shared/exclusive locks. These are discussed in [9]. To our knowledge, no other frameworks for memory consistency models have been generalized in this way.

Acknowledgments. Thanks to many people who have talked with me about memory models, especially Nancy Lynch, Xiaowei Shen, Arvind, David Detlefs and Charles Leiserson.

[1] Hill conjectures that speculative execution will narrow the performance gap between sequential consistency and weaker memory models [6], but the evidence for this is not yet conclusive [10, 4].

1. REFERENCES

[1] S. V. Adve and M. D. Hill. Weak ordering—a new definition. In *Proceedings of ISCA'90*, pages 2–14, May 1990.

[2] M. Frigo and V. Luchangco. Computation-centric memory models. In *Proceedings of SPAA'98*, pages 240–249, June/July 1998.

[3] K. Gharachorloo, D. Lenoski, J. Laudon, P. Gibbons, A. Gupta, and J. Hennessy. Memory consistency and event ordering in scalable shared-memory multiprocessors. In *Proceedings of ISCA'90*, pages 15–26, May 1990.

[4] C. Gniady, B. Falsafi, and T. N. Vijaykumar. Is SC + ILP = RC? In *Proceedings of ISCA'99*, pages 270–278, Jan. 1999.

[5] J. Gosling, B. Joy, G. Steele, and G. Bracha. *The Java Language Specification*. Addison-Wesley, 2nd edition, 2000.

[6] M. D. Hill. Multiprocessors should support simple memory-consistency models. *IEEE Computer*, 31(8):28–34, Aug. 1998.

[7] L. Iftode, J. P. Singh, and K. Li. Scope consistency: A bridge between release consistency and entry consistency. In *Proceedings of SPAA'96*, pages 277–287, June 1996.

[8] L. Lamport. How to make a multiprocessor computer that correctly executes multiprocess programs. *IEEE Transactions on Computers*, C-28(9):690–691, Sept. 1979.

[9] V. Luchangco. *Memory Consistency Models for High Performance Distributed Computing*. PhD thesis, Massachusetts Institute of Technology, June 2001.

[10] P. Ranganathan, V. S. Pai, and S. V. Adve. Using speculative retirement and larger instruction windows to narrow the performance gap between memory consistency models. In *Proceedings of SPAA'97*, pages 199–210, June 1997.

[11] Supercomputing Technologies Group. Cilk 5.3.1. Reference manual, MIT Laboratory for Computer Science, Cambridge, MA, June 1998.