Weak Ordering - A New Definition[†]

Sarita V. Adve
Mark D. Hill

Computer Sciences Department
University of Wisconsin
Madison, Wisconsin 53706

ABSTRACT

A memory model for a shared memory, multiprocessor commonly and often implicitly assumed by programmers is that of *sequential consistency*. This model guarantees that all memory accesses will appear to execute atomically and in program order. An alternative model, *weak ordering*, offers greater performance potential. Weak ordering was first defined by Dubois, Scheurich and Briggs in terms of a set of rules for hardware that have to be made visible to software.

The central hypothesis of this work is that programmers prefer to reason about sequentially consistent memory, rather than having to think about weaker memory, or even write buffers. Following this hypothesis, we re-define weak ordering as a contract between software and hardware. By this contract, software agrees to some formally specified constraints, and hardware agrees to appear sequentially consistent to at least the software that obeys those constraints. We illustrate the power of the new definition with a set of software constraints that forbid data races and an implementation for cache-coherent systems that is not allowed by the old definition.

Key words: shared-memory multiprocessor, sequential consistency, weak ordering.

1. Introduction

This paper is concerned with the programmer's model of memory for a shared memory, MIMD multiprocessor, and its implications on hardware design and performance. A memory model commonly (and often implicitly) assumed by programmers is that of *sequential consistency*, formally defined by Lamport [Lam79] as follows:

> [Hardware is sequentially consistent if] the result of any execution is the same as if the operations of all the processors were executed in some sequential order, and the operations of each individual processor appear in this sequence in the order specified by its program.

Application of the definition requires a specific interpretation of the terms *operations* and *result*. We assume that *operations* refer to memory operations or accesses (e.g., reads and writes) and *result* refers to the union of the values returned by all the read operations in the execution and the final state of memory. With these assumptions, the above definition translates into the following two conditions: (1) all memory accesses appear to execute atomically in some total order, and (2) all memory accesses of each processor appear to execute in an order specified by its program (program order).

Uniprocessor systems offer the model of sequential consistency almost naturally and without much compromise in performance. In multiprocessor systems on the other hand, the conditions for ensuring sequential consistency are not usually as obvious, and almost always involve serious performance trade-offs. For four configurations of shared memory systems (bus-based systems and systems with general interconnection networks, both with and without caches), Figure 1 shows that as potential for parallelism is increased, sequential consistency imposes greater constraints on hardware, thereby limiting performance. The use of many performance enhancing features of uniprocessors, such as write buffers, instruction execution overlap, out-of-order memory accesses and lockup-free caches [Kro81] is heavily restricted.

The problem of maintaining sequential consistency manifests itself when two or more processors interact through memory operations on common vari-

Initially X = Y = 0

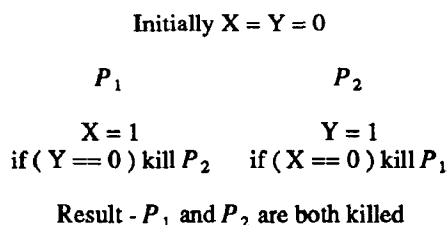| P_1	P_2
X = 1	Y = 1
if (Y == 0) kill P_2	if (X == 0) kill P_1

Result - P_1 and P_2 are both killed

Figure 1. A violation of sequential consistency.

Sequential consistency is violated since there does not exist a total order of memory accesses that is consistent with program order, and kills both P_1 and P_2. Note that there are no data dependencies among the instructions of either processor. Thus simple interlock logic does not preclude the second instruction from being issued before the first in either processor.

Shared-bus systems without caches - The execution is possible if the accesses of a processor are issued out of order, or if reads are allowed to pass writes in write buffers.

Systems with general interconnection networks without caches - The execution is possible even if accesses of a processor are issued in program order, but reach memory modules in a different order [Lam79].

Shared-bus systems with caches - Even with a cache coherence protocol [ArB86], the execution is possible if the accesses of a processor are issued out-of-order, or if reads are allowed to pass writes in write buffers.

Systems with general interconnection networks and caches - The execution is possible even if accesses of a processor are issued and reach memory modules in program order, but do not *complete* in program order. Such a situation can arise if both processors initially have X and Y in their caches, and a processor issues its read before its write is propagated to the cache of the other processor.

ables. In many cases, these interactions can be partitioned into operations that are used to order events, called *synchronization*, and the other more frequent operations that read and write data. If synchronization operations are made recognizable to the hardware, and actions to ensure sequential consistency could be restricted to such operations, then higher overall performance might be achieved by completing normal reads and writes faster. These considerations motivate an alternative programmer's model that relies on synchronization that is visible to the hardware to order memory accesses. Dubois, Scheurich and Briggs have defined such systems in terms of conditions on hardware and have named them *weakly ordered* [DSB86,DSB88, Sch89].

We believe that weak ordering facilitates high performance implementations, but that programmers prefer to reason about sequentially consistent memory rather than weaker memory systems or even write buffers. Hence, a description of memory should not require the specification of the performance enhancing features of the underlying hardware. Rather, such features should be camouflaged by defining the memory model in terms of constraints on software which, if obeyed, make the weaker system appear sequentially consistent.

After surveying related work in Section 2, we give a new definition of weak ordering in Section 3. We illustrate the advantages of this definition with an example set of software constraints in Section 4 and an example implementation in Section 5. Finally, in Section 6, we use these example constraints and example implementation to analyze our framework and compare it with that given by Dubois, et al. For the convenience of the reader, the key definitions used throughout the paper are repeated in Appendix C.

2. Related Work

This section briefly describes relevant previous work on sequential consistency (Section 2.1) and weak ordering (Section 2.2). A more detailed survey of the subject appears in [AdH89].

2.1. Sequential Consistency

Sequential consistency was first defined by Lamport [Lam79], and discussed for shared memory systems with general interconnection networks, but no caches. For single bus cache-based systems, a number of cache-coherence protocols have been proposed in the literature [ArB86]. Most ensure sequential consistency. In particular, Rudolph and Segall have developed two protocols, which they formally prove guarantee sequential consistency [RuS84]. The RP3 [BMW85,PBG85] is a cache-based system, where processor memory communication is via an Omega network, but the management of cache coherence for shared writable variables is entrusted to the software. Sequential consistency is ensured by requiring a process to wait for an acknowledgement from memory for its previous miss on a shared variable before it can issue another access to such a variable. In addition, the RP3 also provides an option by which a process is required to wait for acknowledgements on its outstanding requests only on a *fence* instruction. As will be apparent later, this option functions as a weakly ordered system.

Dubois, Scheurich and Briggs have analyzed the problem of ensuring sequential consistency in systems that allow caching of shared variables, without impos-

ing any constraints on the interconnection network [DSB86, DSB88, ScD87, Sch89]. A sufficient condition for sequential consistency for cache-based systems has been stated [ScD87, Sch89]. The condition is satisfied if all processors issue their accesses in program order, and no access is issued by a processor until its previous accesses have been *globally performed*. A write is globally performed when its modification has been propagated to all processors. A read is globally performed when the value it returns is bound and the write that wrote this value is globally performed.

The notion of *strong ordering* as an equivalent of sequential consistency was defined in [DSB86]. However, there do exist programs that can distinguish between strong ordering and sequential consistency [AdH89] and hence, strong ordering is not strictly equivalent to sequential consistency. Strong ordering has been discarded in [Sch89] in favor of a similar model, viz., concurrent consistency. A concurrently consistent system is defined to behave like a sequentially consistent system for most practical purposes.

Collier has developed a general framework to characterize architectures as sets of rules, where each rule is a restriction on the order of execution of certain memory operations. [Col84, Col90]. He has proved that for most practical purposes, a system where all processors observe all write operations in the same order (called write synchronization), is indistinguishable from a system where all writes are executed atomically.

Shasha and Snir have proposed a software algorithm to ensure sequential consistency [ShS88]. Their scheme statically identifies a minimal set of pairs of accesses within a process, such that delaying the issue of one of the elements in each pair until the other is globally performed guarantees sequential consistency. However, the algorithm depends on detecting conflicting data accesses at compile time and so its success depends on data dependence analysis techniques, which may be quite pessimistic.

The conditions for sequential consistency of memory accesses are analogous to the serialization condition for transactions in concurrent database systems [BeG81, Pap86]. However, database systems seek to serialize the effects of entire transactions, which may be a series of reads and writes while we are concerned with the atomicity of individual reads and writes. While the concept of a transaction may be extended to our case as well and the database algorithms applied, practical reasons limit the feasibility of this application. In particular, since database transactions may involve multiple disk accesses, and hence take much longer than simple memory accesses, database systems can afford to incur a much larger overhead for concurrency control.

2.2. Weak Ordering

Weakly ordered systems depend on explicit, hardware recognizable synchronization operations to order the effects of events initiated by different processors in a system. Dubois, Scheurich and Briggs first defined weak ordering in [DSB86] as follows:

> **Definition 1:** In a multiprocessor system, storage accesses are weakly ordered if (1) accesses to global synchronizing variables are strongly ordered, (2) no access to a synchronizing variable is issued by a processor before all previous global data accesses have been globally performed, and if (3) no access to global data is issued by a processor before a previous access to a synchronizing variable has been globally performed.

It was recognized later in [ScD88, Sch89] that the above three conditions are not necessary to meet the intuitive goals of weak ordering. In Section 3, we give a new definition that we believe formally specifies this intuition.

Bisiani, Nowatzyk and Ravishankar have proposed an algorithm [BNR89] for the implementation of weak ordering on distributed memory systems. Weak ordering is achieved by using timestamps to ensure that a synchronization operation completes only after *all* accesses previously issued by *all* processors in the system are complete. The authors mention that for synchronization operations that require a value to be returned, it is possible to send a tentative value before the operation completes, if a processor can undo subsequent operations that may depend on it, after receiving the actual value. In [AdH89], we discuss how this violates condition 3 of Definition 1, but does not violate the new definition of weak ordering below.

3. Weak Ordering - A New Definition.

We view weak ordering as an interface (or contract) between software and hardware. Three desirable properties for this interface are: (1) it should be formally specified so that separate proofs can be done to ascertain whether software and hardware are correct (i.e., they obey their respective sides of the contract), (2) the programmer's model of hardware should be simple to avoid adding complexity to the already difficult task of parallel programming, and (3) the hardware designer's model of software should facilitate high-performance, parallel implementations.

Let a *synchronization model* be a set of constraints on memory accesses that specify how and when synchronization needs to be done.

Our new definition of weak ordering is as follows.

> **Definition 2:** Hardware is weakly ordered with respect to a synchronization model if and only if it appears sequentially consistent to all software that obey the synchronization model.

This definition of weak ordering addresses the above mentioned properties as follows. (1) It is formally specified. (2) The programmer's model of hardware is kept simple by expressing it in terms of sequential consistency, the most frequently assumed software model of shared memory. Programmers can view hardware as sequentially consistent if they obey the synchronization model. (3) High-performance hardware implementations are facilitated in two ways. First, hardware designers retain maximum flexibility, because requirements are placed on how the hardware should appear, but not on how this appearance should be created. Second, the framework of this definition allows new (and better) synchronization models to be defined as software paradigms and hardware implementation techniques evolve.

However, there are some disadvantages of defining a weakly ordered system in this manner. First, there are useful parallel programmer's models that are not easily expressed in terms of sequential consistency. One such model is used by the designers of asynchronous algorithms [DeM88]. (We expect, however, it will be straightforward to *implement* weakly ordered hardware to obtain reasonable results for asynchronous algorithms.)

Second, for any potential performance benefit over a sequentially consistent system, the synchronization model of a weakly ordered system will usually constrain software to synchronize using operations visible to the hardware. For some algorithms, it may be hard to recognize which operations do synchronization. Furthermore, depending on the implementation, synchronization operations could be much slower than data accesses. We believe, however, that slow synchronization operations coupled with fast reads and writes will yield better performance than the alternative, where hardware must assume all accesses could be used for synchronization (as in [Lam86]).

Third, programmers may wish to debug programs on a weakly ordered system that do not (yet) fully obey the synchronization model. The above definition allows hardware to return random values when the synchronization model is violated. We expect real hardware, however, to be much more well-behaved. Nevertheless, hardware designers may wish to tell programmers exactly what their hardware may do when the synchronization model is violated, or build hardware that offers an additional option of being sequentially consistent for all software, albeit at reduced speed.

Alternatively, programmers may wish that a synchronization model be specified so that it is possible and practical to verify whether a program, or at least an execution of a program, meets the conditions of the model. Achieving this goal may add further constraints to software and hardware.

To demonstrate the utility of our definition, we next give an example synchronization model. In Section 5, we discuss an implementation of a system that is weakly ordered with respect to this synchronization model, but is not allowed by Definition 1.

4. A Synchronization Model: Data-Race-Free-0

In this section, we define a synchronization model that is a simple characterization of programs that forbid data races. We call this model *Data-Race-Free-0* (DRF0) and use it only as an example to illustrate an application of our definition. In Section 6, we indicate how DRF0 may be refined to yield other data-race-free models that impose fewer constraints, are as realistic and reasonable as DRF0, but lead to better implementations. In defining DRF0, we have avoided making any assumptions regarding the particular methods used for synchronization or parallelization. The knowledge of any restrictions on these methods (for example, sharing only through monitors, or parallelism only through do-all loops) can lead to simpler specifications of data-race-free synchronization models.

Intuitively, a synchronization model specifies the operations or primitives that may be used for synchronization, and indicates when there is "enough" synchronization in a program. The only restrictions imposed by DRF0 on synchronization operations are: (1) the operation should be recognizable by hardware, and (2) the operation should access only one memory location. Thus, a synchronization operation could be a special instruction such as a TestAndSet that accesses only a single memory location, or it could be a normal memory access but to some special location known to the hardware. However, an operation that swaps the values of two memory locations cannot be used as a synchronization primitive for DRF0.

To formally specify the second feature of DRF0, viz., an indication of when there is "enough" synchronization in a program, we first define a set of *happens-before* relations for a program. Our definition is closely related to the "happened-before" relation defined by Lamport [Lam78] for message passing systems, and the "approximate temporal order" used by Netzer and Miller [NeM89] for detecting races in shared memory

parallel programs that use semaphores.

A happens-before relation for a program is a partial order defined for an *execution* of the program on an abstract, idealized architecture where all memory accesses are executed atomically and in program order. For such an execution, two operations initiated by different processors are ordered by happens-before only if there exist intervening synchronization operations between them. To define happens-before formally, we first define two other relations, program order or $\xrightarrow{po}$, and synchronization order or $\xrightarrow{so}$. Let op_1 and op_2 be any two memory operations occurring in an execution. Then,

$op_1 \xrightarrow{po} op_2$ *iff* op_1 occurs before op_2 in program order for some process.

$op_1 \xrightarrow{so} op_2$ *iff* op_1 and op_2 are synchronization operations accessing the same location and op_1 completes before op_2 in the execution.

A *happens-before* relation or $\xrightarrow{hb}$ is defined for an execution on the idealized architecture, as the irreflexive transitive closure of $\xrightarrow{po}$ and $\xrightarrow{so}$, i.e., $\xrightarrow{hb}$ $= (\xrightarrow{po} \cup \xrightarrow{so})^+$.

For example, consider the following chain of operations in an execution on the idealized architecture.

$$op(P_1,x) \xrightarrow{po} S(P_1,s) \xrightarrow{so} S(P_2,s) \xrightarrow{po}$$
$$S(P_2,t) \xrightarrow{so} S(P_3,t) \xrightarrow{po} op(P_3,x)$$

$op(P_i,x)$ is a read or a write operation initiated by processor P_i on location x. Similarly, $S(P_j,s)$ is a synchronization operation initiated by processor P_j on location s. The definition of happens-before then implies that $op(P_1,x) \xrightarrow{hb} op(P_3,x)$.

From the above definition, it follows that $\xrightarrow{hb}$ defines a partial order on the accesses of one execution of a program on the idealized architecture. Since, in general, there can be many different such executions of a program (due to the many possible $\xrightarrow{so}$ relations), there may be more than one happens-before relation defined for a program.

To account for the initial state of memory, we assume that before the actual execution of a program, one of the processors executes a (hypothetical) initializing write to every memory location followed by a (hypothetical) synchronization operation to a special location. This is followed by a (hypothetical) synchronization operation to the same location by each of the other processors. The actual execution of the program is assumed to begin after all the synchronization operations are complete. Similarly, to account for the final state of memory, we assume a set of final reads and synchronization operations analogous to the initializing operations for the initial state. Henceforth, an idealized execution will implicitly refer to an execution on the idealized architecture augmented for the initial and final state of memory as above, and a happens-before relation will be assumed to be defined for such an augmented execution.

The happens-before relation can now be used to indicate when there is "enough" synchronization in a program for the synchronization model DRF0. The complete formal definition of DRF0 follows.

Definition 3: A program obeys the synchronization model Data-Race-Free-0 (DRF0), if and only if

(1) all synchronization operations are recognizable by the hardware and each accesses exactly one memory location, and

(2) for any execution on the idealized system (where all memory accesses are executed atomically and in program order), all conflicting accesses are ordered by the happens-before relation corresponding to the execution.

Two accesses are said to *conflict* if they access the same location and they are not both reads. Figures 2a and 2b show executions that respectively obey and violate DRF0.

DRF0 is a formalization that prohibits data races in a program. We believe that this allows for faster hardware than an unconstrained synchronization model, without reducing software flexibility much, since a large majority of programs are already written using explicit synchronization operations and attempt to avoid data races. In addition, although DRF0 specifies synchronization operations in terms of primitives at the level of the hardware, a programmer is free to build and use higher level, more complex synchronization operations. As long as the higher level operations use the primitives appropriately, a program that obeys DRF0 at the higher level will also do so at the level of the hardware primitives. Furthermore, current work is being done on determining when programs are data-race-free, and in locating the races when they are not [NeM89].

5. An Implementation for Weak Ordering w.r.t. DRF0

In the last section, we gave an example synchronization model to illustrate the use of the new definition of weak ordering. This section demonstrates the flexibility afforded to the hardware designer due to the formalization of the synchronization model and the absence of any hardware prescriptions in Definition 2. We give a set of sufficient conditions for implementing weak ordering with respect to DRF0 that allow a viola-
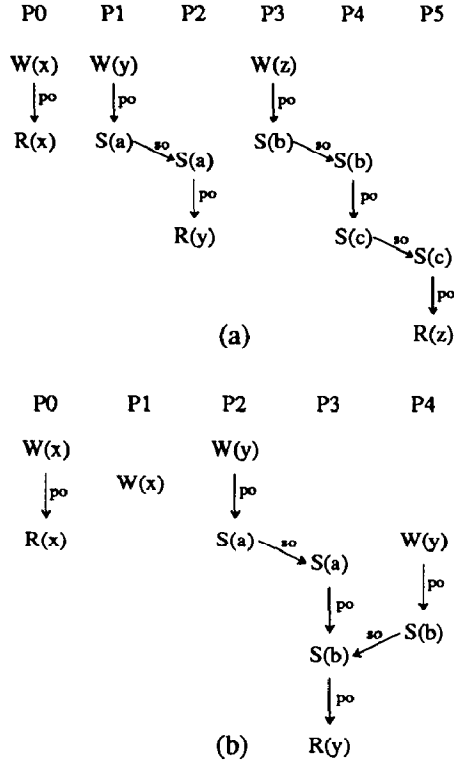
Figure 2. An example and counter-example of DRF0.

Two executions on the idealized architecture are represented. The Pi's denote processors. R(x), W(x) and S(x) respectively denote data read, data write and synchronization operations on the variable x. Time flows downward. An access by processor Pi appears vertically below Pi, in a position reflecting the time at which it was completed. (a) - The execution shown obeys DRF0 since all conflicting accesses are ordered by happens-before. (b) - The execution does not obey DRF0 since the accesses of P0 conflict with the write of P1 but are not ordered with respect to it by happens-before. Similarly, the writes by P2 and P4 conflict, but are unordered.

tion of Definition 1 (Section 5.1). To illustrate an application of these conditions, we describe for a fairly general cache-coherent system (Section 5.2), an example implementation that does not obey the second or the third conditions of Definition 1 (Section 5.3).

5.1. Sufficient Conditions

An implementation based on Definition 1 requires a processor to stall on a synchronization operation until all its previous accesses are globally performed. This serves to ensure that any other processor subsequently synchronizing on the same location will observe the effects of all these accesses. We propose to stall only the processor that issues the *subsequent* synchronization operation until the accesses by the *previous* processor are globally performed. Thus, the first processor is not required to stall and can overlap the completion of its pending accesses with those issued after the synchronization operation. Below, we give a set of sufficient conditions for an implementation based on this notion.

For brevity, we will adopt the following conventions in formalizing our sufficient conditions. Unless mentioned otherwise, *reads* will include data (or ordinary) read operations, read-only synchronization operations, and the read component of synchronization operations that both read and write memory. Similarly, *writes* will include data writes, write-only synchronization operations, and the write component of read-write synchronization operations.

A *commit point* is defined for every operation as follows. A read commits when its return value is dispatched back towards the requesting processor. A write commits when its value could be dispatched for some read. A read-write synchronization operation commits when its read and write components commit. Similarly, a read-write synchronization operation is globally performed when its read and write components are globally performed. We will say that an access is *generated* when it "first comes into existence".

Hardware is weakly ordered with respect to DRF0 if it meets the following requirements.

1. Intra-processor dependencies are preserved.

2. All writes to the *same* location can be totally ordered based on their commit times, and this is the order in which they are observed by all processors.

3. All synchronization operations to the *same* location can be totally ordered based on their commit times, and this is also the order in which they are globally performed. Further, if S_1 and S_2 are synchronization operations and S_1 is committed and globally performed before S_2, then all components of S_1 are committed and globally performed before any in S_2.

4. A new access is not generated by a processor until all its previous synchronization operations (in program order) are committed.

5. Once a synchronization operation S by processor P_i is committed, no other synchronization operations on the *same* location by another processor can commit until after all reads of P_i before S (in program order) are committed and all writes of P_i before S are globally performed.

To prove the correctness of the above conditions, we first prove in Appendix A, a lemma stating a (simpler) *necessary* and *sufficient* condition for weak

7

ordering with respect to DRF0. We then show in Appendix B that the above conditions satisfy the condition of Lemma 1.

The conditions given do not explicitly allow process migration. Re-scheduling of a process on another processor is possible if it can be ensured that before a context switch, all previous reads of the process have returned their values and all previous writes have been globally performed.

5.2. An Implementation Model

This section discusses assumptions for an example underlying system on which an implementation based directly on the conditions of Section 5.1 will be discussed. Consider a system where every processor has an independent cache and processors are connected to memory through a general interconnection network. In particular, no restrictions are placed on the kind of data a cache may contain, nor are any assumptions made regarding the atomicity of any transactions on the interconnection network. A straightforward directory-based, writeback cache coherence protocol, similar to those discussed in [ASH88], is assumed. In particular, for a write miss on a line that is present in *valid* (or *shared*) state in more than one cache, the protocol requires the directory to send messages to invalidate these copies of the line. Our protocol allows the line requested by the write to be forwarded to the requesting processor in parallel with the sending of these invalidations. On receipt of an invalidation, a cache is required to return an acknowledgement (ack) message to the directory (or memory). When the directory (or memory) receives all the acks pertaining to a particular write, it is required to send its ack to the processor cache that issued the write.

We assume that the value of a write issued by processor P_i cannot be dispatched as a return value for a read until the write modifies the copy of the accessed line in P_i's cache. Thus, a write commits only when it modifies the copy of the line in its local cache. However, other copies of the line may not be invalidated.

Within a processor, all dependencies will be assumed to be maintained. Read and write components of a read-write synchronization operation will be assumed to execute atomically with respect to other synchronization operations on the *same* location. All synchronization operations will be treated as write operations by the cache coherence protocol.

5.3. An Implementation

We now outline an example implementation based on the conditions of Section 5.1 for the cache-based system discussed in Section 5.2.

The first condition of Section 5.1 is directly implemented in our model system. Conditions 2 and 3 are ensured by the cache coherence protocol and by the fact that all synchronization operations are treated as writes, and the components of a synchronization operation are executed atomically with respect to other synchronization operations on the *same* location. For condition 4, all operations are generated in program order. In addition, after a synchronization operation, no new accesses are generated until the line accessed is procured by the processor in *exclusive* (or *dirty*) state, and the operation performed on this copy of the line. To meet condition 5, a counter (similar to one used in RP3) that is initialized to zero is associated with every processor, and an extra bit called the *reserve* bit is associated with every cache line. The condition is satisfied as follows.

On a cache miss, the corresponding processor counter is incremented. The counter is decremented on the receipt of a line in response to a read request, or to a write request for a line that was originally in exclusive state in some processor cache. The counter is also decremented when an ack from memory is received indicating that a previous write to a *valid* or *shared* line has been observed by all processors. Thus a positive value on a counter indicates the number of outstanding accesses of the corresponding processor. When a processor generates a synchronization operation, it cannot proceed until it procures the line with the synchronization variable in its cache. If at this time, its counter has a positive value, i.e., there are outstanding accesses, the reserve bit of the cache line with the synchronization variable is set. All reserve bits are reset when the counter reads zero, i.e., when all previous reads have returned their values, and all previous writes have been globally performed[1]. When a processor P_i proceeds after a synchronization operation, it has the exclusive copy of the line with the synchronization variable in its cache. Hence, unless P_i writes back the line, the next request for it will be routed to P_i. When a synchronization request is routed to a processor, it is serviced only if the reserve bit of the requested line is reset, otherwise the request is stalled until the counter reads zero[2]. Condition 5 can be met if it is ensured that a line with its reserve bit set, is never flushed out of a processor cache. A processor that requires such a flush is made to stall until its counter reads zero. However, we believe that such a case will occur fairly rarely and will not be detri-

---

1. This does not require an associative clear. It can be implemented by maintaining a small, fixed table of reserved blocks.

2. This might be accomplished by maintaining a queue of stalled requests to be serviced when the counter reads zero, or a negative ack may be sent to the processor that sent the request, asking it to try again.

mental to performance. Thus for the most part, processors will need to block only to commit synchronization operations[3].

While previous accesses of a processor are pending after a synchronization operation, further accesses to memory will also increment the counter. This implies that a subsequent synchronization operation awaiting completion of the accesses pending before the previous synchronization operation, has to wait for the new accesses as well, before the counter reads zero and it is serviced. This can be avoided by allowing only a limited number of cache misses to be sent to memory while any line is reserved in the cache. This makes sure that the counter will read zero after a bounded number of increments after a synchronization operation is committed. A more dynamic solution involves providing a mechanism to distinguish accesses (and their acks) generated before a particular synchronization operation from those generated after [AdH89].

Though processors can be stalled at various points for unbounded amounts of time, deadlock can never occur. This is because the primary reason a processor blocks is to wait for some set of previously generated data reads to return, or some previously generated data writes and committed synchronization operations to be globally performed. Data read requests always return with their lines. Data writes also all always return with their lines, and their invalidation messages are always acknowledged. Hence, data writes are guaranteed to be globally performed. Similarly, a committed synchronization request only requires its invalidations to be acknowledged before it is globally performed. Since invalidations are always serviced, committed synchronization operations are also always globally performed. Hence a blocked processor will always unblock and termination is guaranteed.

6. Discussion

In this section, we analyze the effectiveness of the new definition for weak ordering (Definition 2) as opposed to the old definition (Definition 1). We perform this analysis by comparing the example hardware implementation (Section 5.3) and the example set of software constraints (DRF0), with the hardware and software allowed by the old definition.

We first claim that the hardware of Definition 1 is weakly ordered by Definition 2 with respect to DRF0. Definition 1 implicitly assumes that intra-processor dependencies are maintained, and that writes to a *given*

location by a *given* processor are observed in the same order by all processors [DSB86]. We assume that condition 1 of Definition 1 requires synchronization operations to be executed in a sequentially consistent manner, and not just strongly ordered. With these additional conditions, our claim can be proved formally in a manner analogous to the proof of Appendix B.

We next determine if the example implementation for the new definition can perform better than an implementation that is also allowed by the old definition. One reason the example implementation may perform better is that with Definition 1 a synchronization operation has global manifestations - before such an operation is issued by a processor, its previous accesses should have been observed by *all* processors in the system. With Definition 2 and DRF0, on the other hand, synchronization operations need only affect the processors that subsequently synchronize on the same location (and additional processors that later synchronize with those processors).

Figure 3 illustrates how the example implementation exploits this difference when two processors, P_0 and P_1, are sharing a data location x, and synchronizing on location s. Assume P_0 writes x, does other work, *Unsets* s, and then does more work. Assume also that after P_0 *Unsets* s, P_1 *TestAndSets* s, does other work and then reads x. Assume further that the write of x takes a long time to be globally performed.
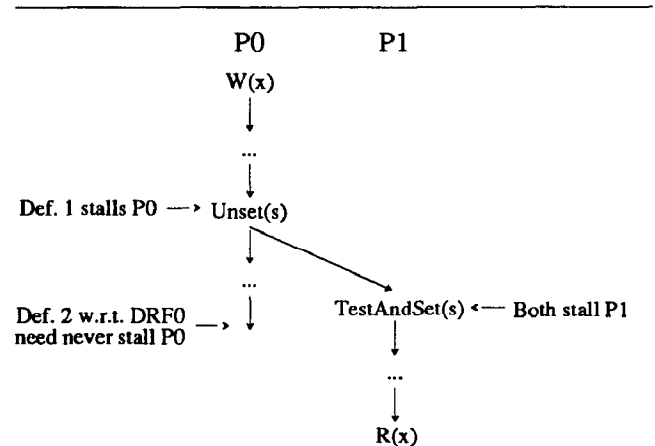

Figure 3. Analysis of the new implementation.

Definition 1 allows P_0 to issue and globally perform data accesses in parallel with the unfinished write of x until it wishes to issue the *Unset* of s. At that time P_0 must stall until the write of x is globally performed. Furthermore, P_1's *TestAndSet* of s cannot succeed until the *Unset* of s, and hence also the write of x, is globally performed.

The example implementation allows P_0 to continue to do other work after it has committed the *Unset* of

---

3. To allow process migration, a processor is also be required to stall on a context switch until its counter reads zero.

s. P_0's further progress is limited only by implementation restrictions, such as, a cache miss that needs to replace the block holding s. P_1's *TestAndSet* of s, however, will still be blocked until P_0's write is globally performed, and *Unset* of s commits. Thus, P_0 but not P_1 gains an advantage from the example implementation.

One very important case where the example implementation is likely to be slower than one for Definition 1 occurs when software performs repeated testing of a synchronization variable (e.g., the *Test* from a *Test-and-TestAndSet* [RuS84] or spinning on a barrier count). The example implementation serializes all these synchronization operations, treating them as writes. This can lead to a significant performance degradation.

The unnecessary serialization can be avoided by improving on DRF0 to yield a new data-race-free model. In particular, a distinction between synchronization operations that only read (e.g., *Test*), only write (e.g., *Unset*), and both read and write (e.g., *TestAndSet*) can be made. Then DRF0 can be modified so that a processor cannot use a read-only synchronization operation to order its previous accesses with respect to subsequent synchronization operations of other processors. This does not compromise on the generality of the software allowed by DRF0 but will allow optimizations that lead to higher performance. In particular, for the example implementation, the read-only synchronization operations need not be serialized, and are not required to stall other processors until the completion of previous accesses.

Finally, we compare the software for which implementations based on Definition 1 appear sequentially consistent, and the software allowed by DRF0. Although the data-race-free model captures a large number of parallel programs, there exist some programs that use certain restricted kinds of data races for which implementations of Definition 1 appear sequentially consistent. Spinning on a barrier count with a data read is one example. We emphasize however, that this feature is not a drawback of Definition 2, but a limitation of DRF0. To allow such races, a new synchronization model can be defined.

7. Conclusions

Most programmers of shared memory systems implicitly assume the model of sequential consistency for the shared memory. This model precludes the use of most performance enhancing features of uniprocessor architectures. We advocate that for better performance, programmers change their assumptions about hardware and use the model of weak ordering, which was originally defined by Dubois, Scheurich and Briggs in terms of certain conditions on hardware. We believe, however, that this definition is unnecessarily restrictive on hardware and does not adequately specify the programmer's model.

We have re-defined weak ordering as a contract between software and hardware where hardware promises to appear sequentially consistent at least to the software that obeys a certain set of constraints which we have called the synchronization model. This definition is analogous to that given by Lamport for sequential consistency in that it only specifies how hardware should *appear* to software. The definition facilitates separate analyses and formal proofs of necessary and sufficient conditions for software and hardware to obey their sides of the contract. It allows programmers to continue reasoning about their programs using the sequential model of memory. Finally, it does not inflict any unnecessary directives on the hardware designer.

To illustrate the advantages of our new definition, we have specified an example synchronization model (DRF0) that forbids data races in a program. We have demonstrated that such a formal specification makes possible an implementation not allowed by the old definition, thereby demonstrating the greater generality of our definition. Finally, we have indicated how some constraints on DRF0 may be relaxed to improve the performance of our implementation.

A promising direction for future research is an application of the new definition to further explore alternative implementations of weak ordering with respect to data-race-free models. A quantitative performance analysis comparing implementations for the old and new definitions of weak ordering would provide useful insight.

Another interesting problem is the construction of other synchronization models optimized for particular software paradigms, such as, sharing only through monitors, or parallelism only from do-all loops, or for specific synchronization primitives offered by specific systems, e.g., QOSB [GVW89]. These optimizations may lead to implementations with higher performance.

9. References

[AdH89]   S. V. ADVE and M. D. HILL, Weak Ordering - A New Definition And Some Implications, Computer Sciences Technical Report #902, University of Wisconsin, Madison, December 1989.

[ASH88]   A. AGARWAL, R. SIMONI, M. HOROWITZ and J. HENNESSY, An Evaluation of Directory Schemes for Cache Coherence, 15th Annual International Symposium on Computer Architecture, Honolulu, Hawaii, June 1988, 280-289.

[ArB86]   J. ARCHIBALD and J. BAER, Cache Coherence Protocols: Evaluation Using a Multiprocessor Simulation Model, ACM Transactions on Computer Systems 4, 4 (November 1986), 273-298.

[BeG81]   P. A. BERNSTEIN and N. GOODMAN, Concurrency Control in Distributed Systems, Computing Surveys 13, 2 (June, 1981), 185-221.

[BNR89]   R. BISIANI, A. NOWATZYK and M. RAVISHANKAR, Coherent Shared Memory on a Distributed Memory Machine, Proc. International Conference on Parallel Processing, August 1989, I-133-141.

[BMW85]   W. C. BRANTLEY, K. P. MCAULIFFE and J. WEISS, RP3 Process-Memory Element, International Conference on Parallel Processing, August 1985, 772-781.

[Col84]   W. W. COLLIER, Architectures for Systems of Parallel Processes, Technical Report Tech. Rep. 00.3253, IBM Corp., Poughkeepsie, N.Y., 27 January 1984.

[Col90]   W. W. COLLIER, Reasoning about Parallel Architectures, Prentice-Hall, Inc., To appear 1990.

[DeM88]   R. DE LEONE and O. L. MANGASARIAN, Asynchronous Parallel Successive Overrelaxation for the Symmetric Linear Complementarity Problem, Mathematical Programming 42(1988), 347-361.

[DSB86]   M. DUBOIS, C. SCHEURICH and F. A. BRIGGS, Memory Access Buffering in Multiprocessors, Proc. Thirteenth Annual International Symposium on Computer Architecture 14, 2 (June 1986), 434-442.

[DSB88]   M. DUBOIS, C. SCHEURICH and F. A. BRIGGS, Synchronization, Coherence, and Event Ordering in Multiprocessors, IEEE Computer 21, 2 (February 1988), 9-21.

[GVW89]   J. R. GOODMAN, M. K. VERNON and P. J. WOEST, Efficient Synchronization Primitives for Large-Scale Cache-Coherent Multiprocessors, Proc. Third International Conference on Architectural Support for Programming Languages and Operating Systems, Boston, April 1989, 64-75.

[Kro81]   D. KROFT, Lockup-Free Instruction Fetch/Prefetch Cache Organization, Proc. Eighth Symposium on Computer Architecture, May 1981, 81-87.

[Lam78]   L. LAMPORT, Time, Clocks, and the Ordering of Events in a Distributed System, Communications of the ACM 21, 7 (July 1978), 558-565.

[Lam79]   L. LAMPORT, How to Make a Multiprocessor Computer That Correctly Executes Multiprocess Programs, IEEE Trans. on Computers C-28, 9 (September 1979), 690-691.

[Lam86]   L. LAMPORT, The Mutual Exclusion Problem, Parts I and II , Journal of the Association of Computing Machinery 33, 2 (April 1986), 313-348.

[NeM89]   R. NETZER and B. MILLER, Detecting Data Races in Parallel Program Executions, Computer Sciences Technical Report #894, University of Wisconsin, Madison, November 1989.

[Pap86]   C. PAPADIMITRIOU, The Theory of Database Concurrency Control, Computer Science Press, Rockville, Maryland 20850, 1986.

[PBG85]   G. F. PFISTER, W. C. BRANTLEY, D. A. GEORGE, S. L. HARVEY, W. J. KLEINFELDER, K. P. MCAULIFFE, E. A. MELTON, V. A. NORTON and J. WEISS, The IBM Research Parallel Processor Prototype (RP3): Introduction and Architecture, International Conference on Parallel Processing, August 1985, 764-771.

[RuS84]   L. RUDOLPH and Z. SEGALL, Dynamic Decentralized Cache Schemes for MIMD Parallel Processors, Proc. Eleventh International Symposium on Computer Architecture, June 1984, 340-347.

[ScD87]   C. SCHEURICH and M. DUBOIS, Correct Memory Operation of Cache-Based Multiprocessors, Proc. Fourteenth Annual International Symposium on Computer Architecture, Pittsburgh, PA, June 1987, 234-243.

[ScD88]   C. SCHEURICH and M. DUBOIS, Concurrent Miss Resolution in Multiprocessor Caches, Proceedings of the 1988 International Conference on Parallel Processing, University Park PA, August, 1988, I-118-125.

[Sch89]   C. E. SCHEURICH, Access Ordering and Coherence in Shared Memory Multiprocessors, Ph.D. Thesis, Department of Computer Engineering, Technical Report CENG 89-19, University of Southern California, May 1989.

[ShS88]   D. SHASHA and M. SNIR, Efficient and Correct Execution of Parallel Programs that Share Memory, ACM Trans. on Programming Languages and Systems 10, 2 (April 1988), 282-312.

Appendix A[4]: A necessary and sufficient condition for weak ordering w.r.t. DRF0.

Lemma 1: A system is weakly ordered with respect to DRF0 if only if for any execution E of a program that obeys DRF0, there exists a happens-before relation $\xrightarrow{hb}$, defined for the program such that (1) every read in E appears in $\xrightarrow{hb}$, (2) every read in $\xrightarrow{hb}$ appears in E, and (3) a read always returns the value written by the last write[5] on the same variable, ordered before it by $\xrightarrow{hb}$.

Proof of necessity

In proving necessity, we do not consider programs for the executions of which an equivalent serialization of accesses can be found as a result of the nature of the initial values of variables and the immediate operands in the code. Instead, we only concern ourselves with general programs where the immediate operands and the initial values may be looked upon as variables which could be assigned arbitrary values and still result in serializable executions.

The proof proceeds by contradiction. Suppose there exists a system which is weakly ordered with respect to DRF0 but does not obey the above condition. Then for any execution E of a program that obeys DRF0, there must be a total ordering T of all its accesses which produces the same result as E, and which is consistent[6] with the program order of all the processes comprising E.

Since T is consistent with program order, there corresponds an execution E' on the idealized architecture that produces the same result as T. Consider the happens-before relation $\xrightarrow{hb}$ corresponding to E'. In E', the partial ordering of accesses as defined by $\xrightarrow{hb}$ is consistent with the order in which accesses are executed. Since all conflicting accesses are ordered by $\xrightarrow{hb}$, they are all executed in the order determined by $\xrightarrow{hb}$. This implies that a read in E' always returns the value of the write ordered last (this is unique for DRF0) before it

by $\xrightarrow{hb}$. Since the result of an execution depends on the value returned by every read, it follows that for E and E' to have the same result, a read in E must appear in $\xrightarrow{hb}$ and vice versa, and a read in E must return the value of the last write ordered before it by $\xrightarrow{hb}$. This contradicts our initial hypothesis. □

Proof of sufficiency

We now prove that a system that obeys the given condition is weakly ordered w.r.t. DRF0. From the given condition, there exists a happens-before relation $\xrightarrow{hb}$ corresponding to any execution E of a program that obeys DRF0, such that every read in E occurs in $\xrightarrow{hb}$ and vice versa, and a read in E returns the value of the write ordered last before it by this happens-before. Consider the execution E' on the idealized architecture, to which this happens-before corresponds.

The order of execution of accesses in E' is consistent with $\xrightarrow{hb}$. Since all conflicting accesses are ordered by $\xrightarrow{hb}$, it follows that a read in E' always returns the value of the write ordered last before it by $\xrightarrow{hb}$. This implies that the result of E is the same as that of E'. Hence it suffices to show that there exists a total ordering of the accesses in E' that is consistent with program order. This is trivially true since E' is an execution on an architecture where all memory accesses are executed atomically and in program order. □

Appendix B: Proof of sufficiency of the conditions in Section 5.1 for weak ordering w.r.t. DRF0

We prove that the conditions of Section 5.1 are sufficient for weak ordering with respect to DRF0 by showing that a system that obeys these conditions also satisfies the necessary and sufficient condition of Lemma 1 in Appendix A.

Consider a program P that obeys DRF0. Let E be an execution of P on a system that obeys the conditions of Section 5.1. Consider the set of accesses A(t) comprising of all the accesses in E that are committed before or at (wall-clock) time t. Define the relations $\xrightarrow{po(t)}$ and $\xrightarrow{so(t)}$ on the accesses in A(t) as follows: $op_1 \xrightarrow{po(t)} op_2$ if and only if op_1 occurs before op_2 in program order for some processor. $op_1 \xrightarrow{so(t)} op_2$ if and only if op_1 and op_2 are synchronization operations that access the same memory location and op_1 commits before op_2. Define $\xrightarrow{xo(t)}$ as the irreflexive, transitive closure of $\xrightarrow{po(t)}$ and $\xrightarrow{so(t)}$. Intuitively, $\xrightarrow{xo(t)}$ reflects the state of the execution E at time t.

---

4. Throughout Appendices A and B, unless mentioned otherwise, reads and writes include synchronization operations. An execution is assumed to be augmented for the initial and final state of memory, as in Section 4.

5. Strictly speaking, with synchronization operations that read and modify memory, sufficiency is guaranteed only if the read of a synchronization operation occurs before its write. Otherwise, the read should be required to return a *modification* of the last write, where the modification depends on the synchronization operation.

6. Two relations A and B are consistent if and only if $A \cup B$ can be extended to a total ordering [ShS88].

---

7. Condition 3 ensures that $\xrightarrow{so}$ defines a total order for all synchronization operations to the *same* location in A(t).

Let the entire execution complete at some time T. We will show that $\xrightarrow{xo(T)}$ is a happens-before relation, and a read in E returns the value of the write ordered last before it by this happens-before relation. Because of the way $\xrightarrow{xo(T)}$ is constructed, every read in E appears in $\xrightarrow{xo(T)}$ and vice versa, and hence, by Lemma 1, the proposition will be proved.

The proof proceeds by contradiction. Suppose the above claim is not true. Then, either $\xrightarrow{xo(T)}$ is not a happens-before relation for P, or else a read in E does not return the value of the last write ordered before it by $\xrightarrow{xo(T)}$. Let t' be the maximum value such that for all $t < t'$, $\xrightarrow{xo(t)}$ could have lead to a $\xrightarrow{xo(T)}$ that is a happens-before relation, and every read that appears in $\xrightarrow{xo(t)}$ returns the value of the last write ordered before it by this happens-before relation. Denote the set of happens-before relations that could have been produced just before time t' as $H(t'-)$.

The time t' is the earliest time at which it can be detected that the system is not sequentially consistent. Hence at t', some data read or some read-only or read-write synchronization operation R, must have committed such that either (i) R does not appear in any of the happens-before relations in $H(t'-)$ or (ii) the value returned by R is from a write that is not ordered last before it by any of the happens-before relations in $H(t'-)$.

We first prove by contradiction that (i) above is not possible. Suppose R does not appear in any of the happens-before relations in $H(t'-)$. The memory accesses generated by a processor are totally governed by the values its reads return. Before t', all the reads that committed returned values that could have lead to some sequentially consistent execution. Hence, these reads could not have lead to the generation of R. Reads that returned values before t' but did not commit are components of read-write synchronization operations. Condition 4 ensures that a processor cannot generate an access until its previous synchronization accesses are committed. Thus, synchronization operations that returned a value but were not committed before t' also cannot result in the generation of R. Thus, we have proved that R does indeed appear in all the happens-before relations in $H(t'-)$.

In the rest of the proof, we show (again by contradiction) that (ii) above is not possible. The argument for (i) also implies that *all* accesses generated before t' appear in all the happens-before relations in $H(t'-)$. Thus $\xrightarrow{xo(t')}$ can lead to at least one of the happens-before relations in $H(t'-)$. Let one of these relations be $\xrightarrow{hb}$. Denote the program order and synchronization order relations corresponding to $\xrightarrow{hb}$ by $\xrightarrow{po}$ and $\xrightarrow{so}$ respectively. Let W' be the write whose value R returns[8]. Since R reads the value written by W', W' must

be committed before or at t'. Hence, W' appears in $\xrightarrow{xo(t')}$ and in $\xrightarrow{hb}$. Therefore, W' is ordered with respect to R by $\xrightarrow{hb}$. But by hypothesis, W' is not the last write ordered before R by $\xrightarrow{hb}$. Let W be the last write ordered before R by $\xrightarrow{hb}$. (DRF0 ensures that this is unique.) Thus, either $W \xrightarrow{hb} R \xrightarrow{hb} W'$ or $W' \xrightarrow{hb} W \xrightarrow{hb} R$. We will prove that R cannot return the value written by W' in either of these cases.

We first prove the following simple results. Below, S_i's are synchronization operations.

(a) If $S_i \xrightarrow{hb} S_j$, and S_j committed before or at t', then S_i committed before S_j.

Proof - If $S_i \xrightarrow{so} S_j$ and S_i did not commit before S_j, then since S_i appears in $\xrightarrow{xo(t')}$, $\xrightarrow{xo(t')}$ cannot be the same as $\xrightarrow{so}$, and hence $\xrightarrow{xo(t')}$ cannot lead to $\xrightarrow{hb}$, a contradiction.

If $S_i \xrightarrow{po} S_j$, and if S_i did not commit before S_j, then either S_i will commit later or S_i will not occur in the execution. The former violates condition 4 and the latter implies that $\xrightarrow{xo(t')}$ cannot lead to $\xrightarrow{hb}$.

Since $\xrightarrow{hb}$ is the transitive closure of $\xrightarrow{so}$ and $\xrightarrow{po}$, it follows that if $S_i \xrightarrow{hb} S_j$, then S_i committed before S_j.

(b) If A is a data access that committed before or at t' and $S_i \xrightarrow{hb} A$, then S_i committed before A was generated.

Proof - Either $S_i \xrightarrow{po} A$ or $S_i \xrightarrow{hb} S_j \xrightarrow{po} A$. Suppose $S_i \xrightarrow{po} A$ and S_i is not committed before A is generated. Then either S_i commits after A is generated, or S_i does not occur in E. The former violates condition 4 and the latter implies that $\xrightarrow{xo(t')}$ cannot lead to $\xrightarrow{hb}$. Hence if $S_i \xrightarrow{po} A$, S_i is committed before t'. Now suppose that $S_i \xrightarrow{hb} S_j \xrightarrow{po} A$. Then from the above argument S_j must have committed before A is generated. Therefore, S_i must have committed before A is generated (result a).

(c) If A_i and A_j are conflicting accesses by different processors such that $A_i \xrightarrow{hb} A_j$, and A_j commits before or at t', then A_i commits before A_j.

If A_i is a synchronization access, then the result follows from results (a) and (b). If A_i is a data access, then either $A_i \xrightarrow{po} S_1 \xrightarrow{so} S_2 \xrightarrow{hb} A_j$ or $A_i \xrightarrow{po} S_1 \xrightarrow{so} A_j$. For the former case, S_2 must commit before A_j is generated (result b), and so S_1 must commit before S_2 (result a), and so A_i must commit before S_2 (condition 5 and because $\xrightarrow{xo(t')}$ can lead to $\xrightarrow{hb}$). Thus A_i commits before A_j is generated, and hence before it is committed. A similar argument can be applied to the latter case.

---

8. We assume that a read always returns the value of some write in the augmented execution.

(d) If A_i and A_j are conflicting data accesses by different processors such that $A_i \xrightarrow{hb} A_j$, A_i is a write operation and A_j commits before or at t', then A_i is globally performed before A_j is generated.

Proof - The argument used in result (c) for the case where A_i and A_j are data accesses applies.

We now use the above results to prove that for each of the cases $W \xrightarrow{hb} R \xrightarrow{hb} W'$, and $W' \xrightarrow{hb} W \xrightarrow{hb} R$, R cannot return the value written by W'.

Case I - $W \xrightarrow{hb} R \xrightarrow{hb} W'$

If $R \xrightarrow{po} W'$, then since condition 1 requires intra-processor dependencies to be maintained, R cannot return the value written by W'.

If R and W' are from different processors, then result (c) requires that R commit before t'. This is a contradiction.

Case II - $W' \xrightarrow{hb} W \xrightarrow{hb} R$

We show that (i) W commits before or at t', and (ii) R cannot return the value of a write that committed before W. From result (c) or from conditions 1 and 2, it will follow that W' commits before W, and hence we will conclude that R cannot return the value written by W'.

Suppose $W \xrightarrow{po} R$. If W is not committed before or at t', then since intra-processor dependencies have to be maintained, it is either known at time t' that W will not be generated, or it is known that W cannot affect the value returned by R. Either of these conditions implies that $\xrightarrow{xo(t')}$ cannot generate $\xrightarrow{hb}$. Thus W must be committed before or at t'. Conditions 1 and 2 then imply that R cannot return the value of a write that committed before W.

If R and W are from different processors, then since W is the last write ordered before R, either both R and W are data operations or both are synchronization operations. If they are both data operations, then from result (d) and condition 2, R cannot return the value of a write committed before W. Result (d) also implies that W is committed before t'.

If R and W are both synchronization operations, then W commits before R (result c), and so R is globally performed after W (condition 3), and so R cannot return the value of a write that committed before W (condition 2).

Thus, we have proved that W always commits before t', and R never returns the value of a write committed before W. It follows that R cannot return the value from W' (result c).

From Case I and Case II, R cannot return the value from W'. This contradicts our hypothesis and completes the proof. $\square$

Appendix C: Glossary of key definitions

[Hardware is sequentially consistent if] the result of any execution is the same as if the operations of all the processors were executed in some sequential order, and the operations of each individual processor appear in this sequence in the order specified by its program.

Definition 1: In a multiprocessor system, storage accesses are weakly ordered if (1) accesses to global synchronizing variables are strongly ordered, (2) no access to a synchronizing variable is issued by a processor before all previous global data accesses have been globally performed and if (3) no access to global data is issued by a processor before a previous access to a synchronizing variable has been globally performed.

Definition 2: Hardware is weakly ordered with respect to a synchronization model if and only if it appears sequentially consistent to all software that obey the synchronization model.

Definition 3: A program obeys the synchronization model Data-Race-Free-0 (DRF0), if and only if

(1) all synchronization operations are recognizable by the hardware and each accesses exactly one memory location, and

(2) for any execution on the idealized system (where all memory accesses are executed atomically and in program order), all conflicting accesses are ordered by the happens-before relation corresponding to the execution.