# WEAK ORDERING - A NEW DEFINITION AND SOME IMPLICATIONS

by

Sarita V. Adve
and
Mark D. Hill

Computer Sciences Technical Report #902

December 1989

# Weak Ordering - A New Definition And Some Implications[†]

*Sarita V. Adve*
*Mark D. Hill*

Computer Sciences Department
University of Wisconsin
Madison, Wisconsin 53706
sarita@cs.wisc.edu
markhill@cs.wisc.edu

A model for correct program behavior commonly and often implicitly assumed by programmers is that of *sequential consistency*, which guarantees that all memory accesses execute atomically and in program order. An alternative programmer's model, *weak ordering*, offers greater performance potential, especially for highly parallel shared memory systems. Weak ordering was first defined by Dubois, Scheurich and Briggs in terms of a set of constraints for hardware, which are not necessary for its intuitive goal.

We define a system to be weakly ordered with respect to a set of software constraints if it appears sequentially consistent to software obeying those constraints. We argue that this definition more directly reflects the intuition behind weak ordering, facilitates a formal statement of the programmer's view of the system, and does not specify unnecessary directives for hardware. For software that does not allow data races, we show that the new definition allows a violation of the old definition and makes possible implementations that have a higher performance potential. We give an example of one such implementation for cache-based systems with general interconnects to illustrate the power of the new definition.

The insight gained from the implementation of weak ordering can also be applied to sequential consistency. We give an algorithm and an implementation for cache-based systems with general interconnects that has a higher performance potential than others we are aware of. We also investigate a markedly new approach that is allowed by our definition to implement weak ordering, and possibly sequential consistency.

[†] The material presented here is based on research supported in part by the National Science Foundation's Presidential Young Investigator and Computer and Computation Research Programs under grants MIPS-8957278 and CCR-8902536, A. T. & T. Bell Laboratories, Digital Equipment Corporation, Texas Instruments, Cray Research and the graduate school at the University of Wisconsin--Madison.

## 1. Introduction

Multiple instruction multiple data (MIMD) machines form an important class of parallel computers. Of these, shared memory systems are particularly attractive, since techniques for programming them are a natural extension of those applied to conventional von Neumann architectures. This paper is primarily concerned with the programmer's model of a shared memory system and its implications on hardware design and performance.

A model for correct behavior of programs commonly (and often implicitly) assumed by programmers is that of *sequential consistency*, formally defined by Lamport [Lam79] as follows:

> [A system 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.

While the definition leaves the specific interpretation of the term *operations* undefined, for a shared memory system, it is usually assumed to refer to memory operations or accesses (e.g., reads and writes). Thus, stated simply for a shared memory system, 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 a single process appear to execute in program order.

Uniprocessor systems offer the model of sequential consistency almost naturally and without much compromise in performance. In the simplest of architectures, where a processor is allowed to issue a memory access only after the previous access in program order is complete, a total order of memory accesses can be obtained based on the wall-clock time of their issue or execution. More sophisticated architectures allow overlap of instruction execution, out-of-order memory accesses, write buffers, caches (which may be lock-up free [Kro81]), etc. In these machines, an ordering of memory accesses based on wall-clock time of issue or execution may violate program order, but interlock logic assures that accesses *appear* to execute in program order, thereby maintaining sequential consistency[1]. Virtual memory and multiprogramming on uniprocessors are also compatible with sequential consistency. If before every context switch, all pending accesses of the currently running process are completed, a total ordering of accesses can be obtained.

In shared memory, multiprocessor systems, 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.

1. An exception to this are architectures that allow imprecise interrupts [AST67], thus violating sequential consistency on those interrupts, in favor of high performance [SmP88]. However, this violation does not have any significant effects on the programmer's model.

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 dependencies between the instructions executed by either $P_1$ or $P_2$. Thus simple interlock logic does not preclude the second instruction from being issued before the first for each of the processors.

**Shared bus based 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 based systems with caches** - A cache coherency protocol is required [ArB86]. Even so, 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. Assume initially $P_1$ and $P_2$ both had X and Y in their caches with a value of 0. It is possible for $P_1$ to issue its read for Y before the write to Y executed by $P_2$ propagates to $P_1$'s cache. Similarly, it is also possible for the write to X issued by $P_1$ to not propagate to $P_2$ before it reads X. Thus both $P_1$ and $P_2$ can read 0 for X and Y and kill each other. This situation arises because accesses of a processor do not *complete* in program order. Note that a cache coherency protocol does not preclude this possibility[2]. Correct behavior can be achieved if a processor does not issue an access until values read or written by its previous accesses (in program order) have been observed by the rest of the processors in the system [ScD87].

The problem of maintaining sequential consistency manifests itself when two or more processors interact through memory operations on common variables. In many cases, these processor 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 only such operations, then higher overall performance might be achieved by completing normal reads and writes faster. Additionally, an optimizing compiler might have more flexibility in rearranging code between synchronization operations.

The above factors motivate an alternative programmer's model which relies on synchronization that is visible to the hardware to order memory accesses. Dubois, Scheurich and Briggs have discussed such systems in [DSB86, DSB88, Sch89] and have called them *weakly ordered*. Intuitively, a weakly ordered system is one that

2. In the most general case, a cache-coherency protocol merely ensures that all write operations to any *given* memory location are observed by all processors in the system in the same order [Arc87]. Note that this is different from the definition of sequential consistency which requires all writes to *all* memory locations to be observed in the same order [Sch89].

appears to be sequentially consistent if programs always obey a certain set of constraints. Dubois et al. have formalized weak ordering in terms of certain conditions on hardware, which we shall discuss later.

In this paper, after first surveying related previous work (Section 2), we give a definition of weak ordering entirely in terms of what it offers to software, based on the above mentioned intuition, rather than as a prescription for hardware (Section 3.1). This is analogous to the definition given by Lamport for sequential consistency. We identify an important case to which we apply the new definition (Section 3.2). We demonstrate the power of the new definition by giving an algorithm for implementing weak ordering, that violates the constraints of the old definition (Section 4.1). For a cache-based system with general interconnects (System 4.2), we give an implementation for the algorithm (Section 4.3) and qualitatively show that this implementation has a higher performance potential than others based on the old definition (Section 4.4). The insight gained by this implementation also motivates a new implementation for sequential consistency (Section 5). To further illustrate the generality and applicability of the new definition, we conclude by discussing a markedly different approach for implementing weak ordering, and possibly sequential consistency (Section 6).

We believe that the problem of ensuring sequential consistency or weak ordering is sufficiently complex that any algorithm to implement these should be supported with a corresponding proof of correctness. Hence proofs of all stated results and algorithms are either provided in the Appendices at the end of the paper or outlined in the relevant sections. For the reader's convenience, Appendix F repeats the key definitions and lemma used throughout the paper.

## 2. Related Work

This section summarizes, and comments on the significance of previous work on ensuring "correct" execution of programs on various classes of parallel systems. Section 2.1 deals primarily with sequential consistency, while Section 2.2 concerns weak ordering. Though a large amount of literature exists on this subject, to our knowledge, no such survey has been published so far.

### 2.1. Sequential consistency

As discussed in the introduction, sequential consistency was first defined by Lamport [Lam79], and discussed for shared memory systems with a general interconnection network, but without any caches. Lamport shows that sequential consistency can be guaranteed in such a system if (1) all accesses in a process are issued in program order, (2) an access is not issued by a process until the previous access has been queued by the corresponding memory module, and (3) a memory module services all accesses to a given location in FIFO order.

For cache-based systems, where processors are connected to memory modules through a common bus, 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 proofs rely on the presence of a single shared bus. However, to increase the number of processors that can be efficiently utilized with these protocols, they propose the use of multiple buses which can together be *logically* treated as one single bus. Memory is divided into distinct modules and each module is connected to all processors through exactly one bus.

The RP3 system built at IBM, in cooperation with the Ultracomputer project from NYU, is a cache-based system, where processor memory communication is via an Omega network [BMW85, GGK83, PBG85]. However, in this system, the management of cache coherency for shared, writable variables is entrusted to the software. Thus from the viewpoint of the hardware, the RP3 is similar to the system studied by Lamport in [Lam79]. The second condition given by Lamport is met in the RP3, by requiring a process to wait for an acknowledgement from memory for its previous miss on a shared writable 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 its acknowledgements on its outstanding requests only on a *fence* instruction. As will be apparent later, this option allows the RP3 to function as a weakly ordered system.

Dubois, Scheurich and Briggs have analyzed the problem for systems that allow caching of shared variables, without imposing any constraints on the interconnection network [DSB86, DSB88, ScD87, Sch89]. They have introduced and defined *strong* and *weak* ordering as two models of shared memory system behavior [DSB86]. Strong ordering essentially requires that when a processor $i$ observes a write operation by some other processor $k$, it also observes all accesses that were observed by $k$ before it issued the write. In addition, accesses issued by a single processor are also required to be initiated, issued and performed (refer [DSB86] for the distinction between these terms) in program order. It was claimed earlier that a system that is strongly ordered is also sequentially consistent [DSB86, DSB88, ScD87, ScD88]. However, Figure 2 shows an execution that is not precluded by the above conditions for strong ordering, but violates sequential consistency. It has been confirmed by Dubois and Scheurich that strong ordering is *not equivalent* to sequential consistency [DuS89].

In [Sch89], strong ordering was discarded in favor of a similar model called concurrent consistency. The sufficient conditions to ensure concurrent consistency given in [Sch89] can be seen to be similar to those discussed above for strong ordering. The formal definition states that a system is concurrently consistent if it executes all programs in the way that sequentially consistent systems do except for programs which explicitly test for sequential consistency or take access timings into consideration. In other words, while a concurrently consistent system appears sequentially consistent for most meaningful synchronization algorithms, there do exist programs for which such a system may yield results not allowed by sequentially consistent systems. Figure 2 shows an

Initially X = Y = 0 in processor caches

| $P_1$ | $P_2$ | $P_3$ |
|---|---|---|
| X = 1 | | |
| | $x_2$ = X | |
| | $y_2$ = Y | |
| | | Y = 1 |
| | | $x_3$ = X |

Result - $x_2 = 1, y_2 = 0, x_3 = 0$

Figure 2. A violation of sequential consistency in a strongly ordered (or concurrently consistent) system.

Consider a cache based system with a general interconnect. X and Y are shared variables initially present in processor caches with value 0. $x_2$ and $y_2$ are local variables, possibly registers, belonging to $P_2$. Similarly, $x_3$ is local to $P_3$. Let all the accesses be performed in program order. It is possible for the write on X to be propagated to $P_2$ before $P_3$ (the invalidation for X could reach $P_2$ before it reached $P_3$). It is also possible for the write on X to be propagated to $P_2$ before the write on Y. Thus reads issued by $P_2$ could return a 1 for X and a 0 for Y, making it appear that X was written before Y. $P_3$ on the other hand can still return a 0 for X, making it appear that Y was written before X. Therefore, there does not exist any total ordering of memory accesses for this execution and hence the system is not sequentially consistent. However, none of the conditions for strong ordering (or concurrent consistency) are violated.

execution of one such program. Note however that the set of accesses in this execution do not appear to have been obtained from any of the commonly used synchronization algorithms. One drawback, however, of this model of "almost sequential consistency" is that it does not give a formal specification of the software for which a system obeying this model will appear sequentially consistent.

Along with concurrent consistency, the models of sequential consistency and weak ordering were identified in [Sch89] for shared memory systems. The work on weak ordering is the topic of the next subsection. For sequential consistency in cache-based systems with general interconnects, a sufficient condition has been proposed. 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 operation is said to be globally performed when a subsequent read to the same location issued by any processor is guaranteed to return the value of this or a later write ("later" here is determined by the order in which the values of the writes become readable by the processor). A read operation is globally performed when the value it returns is fixed, and the write corresponding to this value is globally performed. Thus after a read R is globally performed, no subsequent read to the same location by any processor can return a value older than that returned by R. In subsequent sections, an access will be said to be *complete* only if it is globally performed.

Collier has developed a general framework to characterize architectures in terms of the ordering of events that they permit [Col84a, Col84b, Col85, Col90]. An architecture is defined as a set of rules, where each rule is a restriction on the order of execution of certain events. Collier has proved that for most practical purposes, a system where all processors observe the effects of all write operations in the same order (called the rule of write

synchronization), is indistinguishable from a system where all writes are executed atomically (the rule of write atomicity). Thus it follows that for sequential consistency, the condition that writes should be atomic, can be replaced by that of write synchronization. Collier has similarly proved equivalence (and inequivalence) of various other sets of rules, and hence of the architectures defined by them. However, no implementations of any of these rules have been given for cache-based systems with general interconnects. Nicholas [Nic89] has identified several possible orderings of memory accesses in shared memory systems, and characterized them in terms of the architectural rules defined by Collier. A hierarchy of systems based on the orderings of events that they permit has been constructed.

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 of each pair until the completion of the other, guarantees sequential consistency. Recall that particularly for cache-based systems, "completion" is to be interpreted as "globally performed". The conditions given by Dubois and Scheurich for sequential consistency in cache-based systems [ScD87, Sch89], require a process to always wait for the previous access to complete before the next access can be issued. The algorithm given by Shasha and Snir finds the minimum number of pairs of accesses between which such a delay is required. 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. This can have a large negative impact on the performance of the algorithm. Furthermore, as presented, the algorithm assumes a straightline code with no branch or jump instructions. Extensions to accommodate these are given, but may result in an even more pessimistic analysis.

There have been other approaches that partially or completely rely on software to make parallel executions appear sequential. Jefferson first introduced the concept of virtual time and the time warp mechanism in [Jef85] for correct synchronization of distributed message passing systems. Knight [Kni86], and Tinker and Katz [TiK88] have described schemes for parallelizing mostly functional languages with a few side-effects. Each of these approaches schedules parallel tasks optimistically, without any attention to serializability. The runtime environment detects if any execution could preclude serialization, in which case the relevant process is rolled back to a consistent state. However, it is not clear that the potential speed-up with these schemes justifies the additional overhead.

The conditions for sequential consistency of memory accesses are analogous to the serialization condition for transactions in concurrent database systems. Hence, it may appear that sequential consistency of memory accesses can be ensured by the algorithms proposed for database systems [BeG81, Pap86]. However, there are some major differences which make this approach inefficient. Database systems seek to serialize the effects of entire transactions, which may be a series of reads and writes, while we are only 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 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. Thus, while optimistic methods work well for database systems, it is not clear that the additional overhead incurred is justifiable for our case. Another important difference is that database concurrency control software cannot make a priori assumptions regarding data that will or will not be accessed by two concurrent transactions. Hence worst case behavior has to be assumed. As opposed to this, in our case, a programmer can ascertain which locations are potentially accessible by two processes running concurrently. This can lead to certain optimizations and in fact, forms the basis of our definition of weak ordering. We will see that though the algorithms we give for weak ordering require some kind of "locking", as in the two phase locking protocols of database systems, the above mentioned property makes them far more efficient and less restrictive. Finally, database systems do not allow processors to maintain private copies of data across transactions, as is possible in systems with registers. This makes an extension of database algorithms to sequential consistency of memory operations non-trivial.

So far, we have been interpreting the term "operations" in Lamport's definition of sequential consistency to mean "memory accesses". However, it is possible for a program to require atomicity at a higher level of granularity (e.g., processor instructions, or an aggregate of instructions). Shasha and Snir [ShS88] show that in general, atomicity of multiple memory accesses cannot be ensured just by introducing delays between accesses of a process. They propose the use of an explicit *lock* instruction and give the minimum such locks that need to be used. Nicholas also explores this concept [Nic89] and concludes that explicit synchronization is required in general, for multiple memory accesses to be atomic. For this reason, programs that require atomicity of multiple accesses for correctness, may be best executed on weakly ordered systems. Henceforth, we will always interpret sequential consistency as implying atomicity for individual memory accesses.

In [Lam86b], Lamport has developed a formalism for proving the correctness of programs, without making any assumptions regarding the atomicity of memory accesses. However, it is assumed that an access is not issued until the previous access of the same process is complete. Thus, this model also imposes substantial constraints on the hardware of shared memory multiprocessor systems. In the next subsection, we discuss the model of weak ordering, which considerably relaxes the constraints imposed by sequential consistency.

### 2.2. Weak ordering

As discussed in the introduction, 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 this key idea 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.

Intuitively, the definition states that if shared variables are always accessed in critical sections, a weakly ordered system appears to be sequentially consistent. It was recognized later in [ScD88, Sch89] that the above three conditions are not necessary for meeting this intuitive definition. In Section 3.1, we give a new definition that is closer to the intuitive goal.

Bisiani, Nowatzyk and Ravishankar have proposed an algorithm [BNR89] for the implementation of weak ordering on distributed memory systems. The algorithm depends on assigning timestamps to every processor request on its creation. Processors communicate with each other to determine approximately the oldest message in the system. The period extending from the timestamp of this message is called the Gray Zone. While ordinary reads and writes are serviced immediately, synchronization operations are serviced only when they become older than the current Gray Zone. This achieves weak ordering by ensuring that a synchronization operation completes only after *all* accesses previously issued by *all* processors in the system are complete. The authors mention that it is possible to send a tentative value for a synchronizing variable before a synchronization operation completes, if a process can undo subsequent operations that may depend on it, after receiving the actual value. Note that this violates condition 3 of Definition 1, but as we shall see later, does not violate the new definition of weak ordering in Section 3.1. A major disadvantage of this algorithm is that it relies on the presence of a global clock to assign timestamps. Hence, its scalability is questionable. Furthermore, no performance measures have been given comparing this scheme with the acknowledgement-based approach as in the RP3. Particularly with respect to average latency of synchronization operations, it is not clear that the given scheme is superior.

## 3. Weak ordering - a new definition.

In Section 2.2, we discussed Definition 1 of weak ordering. We noted that the conditions imposed by it are not necessary for the intuition behind weak ordering. Furthermore, Definition 1 is more of a prescription on hardware, and by itself does not clearly specify a system model for software. We give a new definition of weak ordering so as to overcome these drawbacks. This definition allows a violation of the constraints of Definition 1, to yield implementations with higher performance potential. In particular, 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. The new definition enforces constraints only for processors that subsequently synchronize on the same location.

### 3.1. A definition of weak ordering

Let us first recall the motivation for weak ordering. In Section 1, we informally demonstrated how the implementation of sequential consistency in highly parallel systems precludes the incorporation of many common performance enhancing features from uniprocessor architectures. Sequential consistency also places restrictions on optimizations that can be performed by a compiler. These factors motivated the need for a new programming model that has been called weak ordering. Weak ordering is essentially a trade-off between software flexibility and hardware performance. Intuitively, a weakly ordered system promises to give to programmers their model of sequential consistency only if programs obey certain constraints. Thus, a system may or may not be weakly ordered *with respect to a certain set of software constraints.* As mentioned in the introduction, the operations that are key to consistent behavior are the synchronization operations. Hence, the constraints on software for which a system may appear sequentially consistent revolve around these synchronization operations. We call such constraints on synchronization as the *synchronization model*, and elaborate on how to specify them formally later. We define a system to be weakly ordered with respect to *a* synchronization model as follows.

> **Definition 2:** A system is weakly ordered with respect to a synchronization model if and only if it appears sequentially consistent to all executions of a program that obey the synchronization model.

A system appears sequentially consistent to an execution, if all memory accesses of the execution can be totally ordered in a manner consistent with program order. The term "program" in the above definition refers to any collection of software running on the system.

The new definition for weak ordering is analogous to that given by Lamport for sequential consistency in that it is purely from the programmer's point of view, and it does not specify any explicit directives for hardware.

However, there are some disadvantages of defining a weakly ordered system in this manner. First, the operation of algorithms that are inherently *asynchronous* and do not rely on sequential consistency for correctness [DeM88] is left unspecified. This disadvantage is easily handled by *implementing* weakly ordered systems so that for such algorithms, reasonable results are obtained.

Second, programmers may wish to debug programs on a weakly ordered system that do not (yet) fully obey the synchronization model. One approach to addressing this is to build hardware that offers an additional option of being sequentially consistent for all programs, albeit at reduced speed.

Third, for any potential performance benefit over a sequentially consistent system, a synchronization model for a weakly ordered system will usually constrain software to synchronize using operations recognizable by the hardware. Depending on the implementation, these operations can themselves result in some performance penalty. However, we believe 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 (e.g., in [Lam86a, Lam87]).

Finally, 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. This may involve a trade-off between the allowable software and computational complexity of the verification algorithm. Database scheduling algorithms make a similar compromise by allowing only conflict serializable as opposed to the more general view serializable schedules for computational ease [Pap86].

In the next section, we identify a synchronization model that we believe offers a good trade-off between hardware and software flexibility, at least within the domain of currently existing architectures and programming styles.

### 3.2. A synchronization model: Data-Race-Free

We identify a synchronization model that we call "Data-Race-Free" (DRF). Intuitively, a synchronization model specifies the primitives that may be used for synchronization, and indicates when there is "enough" synchronization in a program. DRF does not place any restrictions on primitives that may be used for synchronization, as long as they are recognizable by the hardware. For example, such a primitive could be a special instruction like a TestAndSet, or it could be a normal memory access but to some special location known to the hardware.

To formally specify the second feature of DRF, 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 issued on 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 $\xrightarrow{so}$. Thus, $\xrightarrow{hb} = ( \xrightarrow{po} \cup \xrightarrow{so} )^+$, for an execution on the idealized architecture.

For example, consider the following chain of operations in the execution.

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

$op(P_i,x)$ 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 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.

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

**Definition 3:** A program obeys the synchronization model Data-Race-Free (DRF), if and only if
(1) all synchronization operations are recognizable by the hardware, 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 assumed to *conflict* if they are to the same location and not both are reads. Figures 3a and 3b show executions that respectively obey and violate DRF.

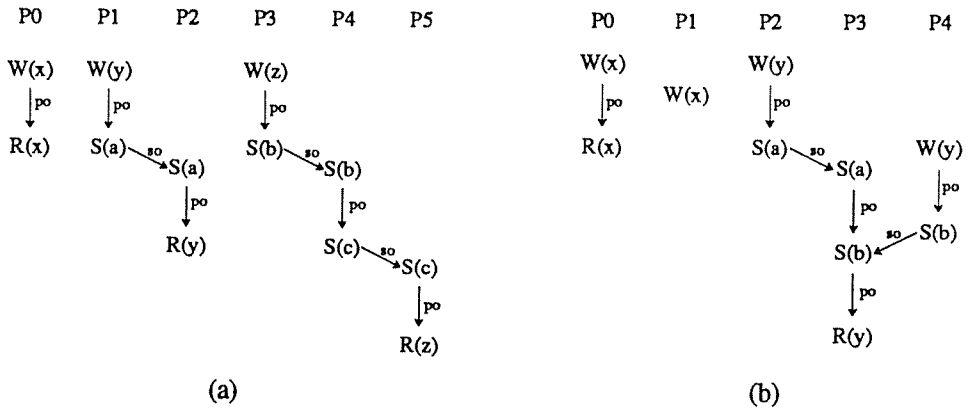


Figure 2. Examples to illustrate the synchronization model DRF.

Two executions on the idealized architecture are represented. The Pi's denote processors. R(x), W(x) and S(x) respectively denote read, write and synchronization operations on the variable x. Time is assumed to increase from top to bottom in the vertical direction. An access issued by processor Pi appears vertically below Pi, in a position reflecting the time it was executed relative to other accesses. Part (a) - The execution shown obeys DRF since all conflicting accesses are ordered by happens-before. Part (b) - The execution does not obey DRF since the accesses of P0 conflict with the write of P1 but are not ordered w.r.t. it by happens-before. Similarly the writes by P2 and P4 conflict, but are unordered.

We believe that DRF allows for faster hardware without significantly reducing software flexibility. Since the use of normal reads and writes for synchronization can result in complex and hard to debug programs, a large majority of programs are already written using explicit synchronization operations. In addition, though DRF 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 DRF 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 on debugging executions that are not [NeM89].

Note that since synchronization operations may now be different from normal reads and writes, the term *operations* in the definition of sequential consistency should be extended to include these synchronization operations as well.

## 4. The power of the new definition with DRF

In the last section, we defined weak ordering with respect to *a* synchronization model (Definition 2) and identified one synchronization model (DRF) that we believe offers a good trade-off between software and hardware. We now argue that Definition 2 with DRF provides a more powerful definition of weak ordering than does Definition 1, because it allows (a) practically any useful program execution (software) allowed by Definition 1, (b) any implementation (hardware) allowed by Definition 1, and (c) at least one (practical) implementation *not* allowed by Definition 1.

Definition 1 does not formally define the allowed program executions. We cannot conceive of any useful software that may be allowed by Definition 1, but not by DRF. Intuitively, both software models require inter-processor data conflicts to be ordered by intervening synchronization operations.

Definition 2 with DRF allows all implementations allowed by Definition 1. Though this result is fairly intuitive, we give a formal proof to substantiate the intuition. In Appendix A, we prove a lemma stating the necessary and sufficient condition for weak ordering w.r.t. DRF. We then show in Appendix B that Definition 1 satisfies this condition.

In the rest of this section, we give an algorithm (Section 4.1) for implementing weak ordering w.r.t. DRF that violates the constraints imposed by Definition 1. For a fairly general cache-based system (Section 4.2), we describe an implementation for this algorithm (Section 4.3). We show qualitatively that the implementation has a higher performance potential than those based on Definition 1 (Section 4.4). Though we do not claim that this implementation is optimal, we do believe that it is practical and adequately demonstrates the additional power of Definition 2 over Definition 1.

### 4.1. An aggressive algorithm for weak ordering w.r.t. DRF

This section discusses an algorithm that satisfies Definition 2 with DRF, but violates Definition 1. The latter definition requires a processor to stall on a synchronization operation until all previous accesses issued by it are completed. 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 complete. This can lead to higher performance by preventing the first processor from ever stalling. The complete algorithm follows.

*Algorithm*

A system is weakly ordered w.r.t. DRF if it meets the following requirements.

1. All processors observe writes on a given location in the same order and dependencies within a processor are preserved.
2. Synchronization operations to the same location are totally ordered, i.e., appear atomic.
3. An access to a global variable is not issued by a processor until all its previous synchronization operations (in program order) are complete.
4. After a synchronization operation S, has been issued by $P_i$, no other synchronization operations on the same location by a processor other than $P_i$ are serviced until all accesses of $P_i$ before S (in program order) are complete.

The proof of correctness for the above algorithm, which proceeds in a manner similar to the proof in Appendix B, is presented in Appendix C.

The algorithm as stated, applies to any general system. For cache-based systems in particular, requirement 3 above can be relaxed. In such systems, it is not necessary for the previous synchronization operations to *complete*. As long as the operation has been successfully performed on its copy of the line, a processor can proceed with its next global access even if the synchronization operation has not yet been observed by *other* processors. The proof is only a slight modification of the above, as indicated in Appendix C.

### 4.2. Implementation model

This section discusses assumptions for an example underlying system on which an implementation of the new algorithm given in the previous section will be discussed. Note that these assumptions are not necessary for the algorithm and are merely made to demonstrate the feasibility of an implementation on a realistic, highly parallel, shared memory system.

We consider a system where every processor has an independent cache and processors are connected to memory through a general interconnection network. 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, write-back cache coherency protocol, similar to those discussed in [ASH88], is assumed. The protocol ensures that every processor observes writes on a *given* location in the same order. The protocol we assume involves two states for lines in memory, viz., valid and modified; and three for lines in a cache, viz., read-only (RO), read/write (RW) and invalid. A line in memory is said to be in the *valid* state if it is a currently valid copy of the line. While a line is in valid state in memory, it can either be invalid or in RO state in one or more processor caches. A line in *modified* state in memory exists in exactly one processor cache, in the RW state. This is the only valid copy of the line in the entire system. A line which is neither RO nor RW in a processor cache is *invalid*.

A read request issued by a processor is satisfied by its cache if it contains the line in either of RO or RW states. If the line is invalid in the cache, the request is sent to the relevant memory module (assuming that the directory is distributed across the memory modules). If memory has the line in valid state, it sends it to the processor. If it does not, the request is routed to the processor which has the line in RW state. In either case, the line is finally stored in the requesting processor's cache as RO. In the latter case, the processor with the modified line converts its state in its cache to RO and sends a copy to memory as well, to be kept there as valid.

A write request by a processor is satisfied by its cache if it contains the line in RW state. Otherwise, as for the read, the line is either sent by memory if it has the line in valid state, or by the processor with the only copy of the modified line. This processor is then required to invalidate the line in its cache. In either case the requesting processor stores the line in its cache as RW and memory marks the line modified.

On any cache miss, if the line in the cache frame to be allocated to the new request is in the RW state, it is written back to memory. This operation is referred to as a *write-back* or a *flush*.

On a write miss that is present in RO state in more than one cache, the memory (or directory) is required to send a message to these caches to invalidate the line. This invalidation message may be a broadcast or a multicast depending on the size of the directory [ASH88]. Our protocol allows the line requested by the write to be forwarded along with these invalidation messages. On receipt of an invalidation, a cache is required to return an acknowledgement (ack) message to the memory. When the 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. The acks are not required for cache coherency, but are used for maintaining weak ordering, as we shall see later.

For the sake of simplicity, it will be assumed that there is no process migration. However, a process can be easily re-scheduled on a different processor by ensuring that before a context switch all pending accesses of the current process are complete. With no process migration, the coherency protocol described is clearly required only for writable variables which are shared among various processors. Private data and instruction code may be handled separately.

In addition to all the above, it will be assumed that within a processor, data dependencies are not violated. No assumptions are made regarding the nature of the synchronization operations except that all such operations to the same location *appear* to be performed atomically with respect to each other.

### 4.3. An aggressive implementation for weak ordering w.r.t. DRF

To demonstrate the feasibility of the algorithm of Section 4.1, we now outline an implementation on the cache-based system discussed in the previous sub-section. The first two requirements of our algorithm are directly implemented in our model system. For the third requirement, all operations are issued in program order. Also, on a synchronization operation, no further accesses are issued until the line with the synchronizing variable is procured by the processor and the operation performed on this copy of the line[3]. To meet requirement 4, as shown in Figure 3, a counter that is initialized to zero is associated with every processor (similar to RP3), and an extra bit called the *reserve* bit is associated with every cache line. The requirement is satisfied as follows.

On every cache miss, the corresponding processor counter is incremented. For a read miss, and a write miss to a line in modified state in memory, the counter is decremented when the line requested is returned. For a write miss to a line in valid state, the counter is decremented when the acknowledgement for the write is received from memory. Thus a positive value on a counter indicates the number of incomplete accesses of the corresponding processor. When a processor issues 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 reaches zero, i.e., there are no pending accesses. Note that when a processor proceeds after a synchronization operation, it has the synchronization variable in RW state in its cache. Hence the next request for this variable will be routed to this processor for service. When a request for a synchronization variable is routed to a processor, it is serviced only if the reserve bit is reset, otherwise the request is stalled until the counter reads zero[4]. Now requirement 4 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 previously pending accesses are

3. This actually satisfies the relaxation on requirement 3 discussed earlier.

4. 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 issued the request, asking it to try again.

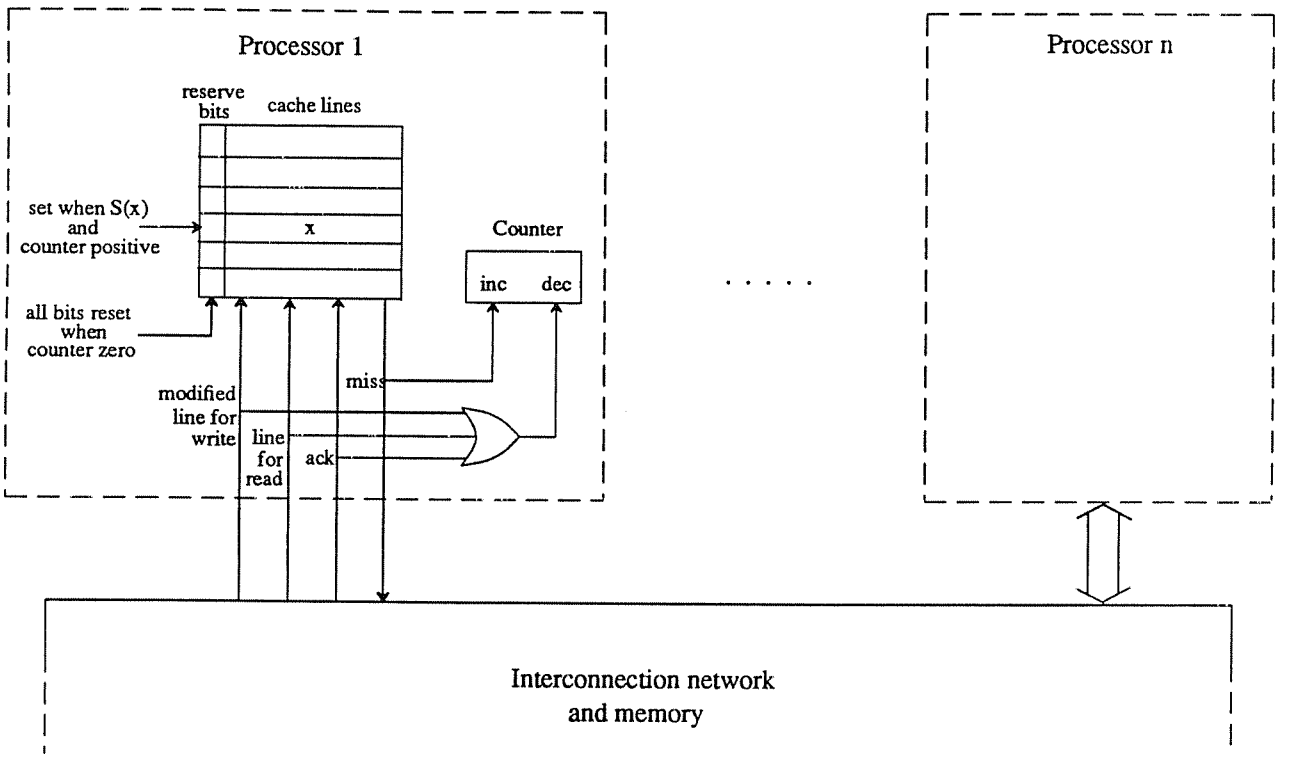


Figure 3. An implementation of weak ordering.

complete (indicated by the counter reading zero). However, we believe that such a case will occur fairly rarely and will not be detrimental to performance. Thus for the most part, processors will need to block only to complete synchronization operations.

Note that 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 to complete 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 issued. A more dynamic solution involves providing a mechanism to distinguish accesses (and their acks) issued before a particular synchronization operation from those issued after. We can alternate between "type-1" and "type-2" classes of accesses and use two counters for each processor. These are independently updated due to accesses and acks of the corresponding types. A stalled synchronization operation now need just wait until the counter associated with the type of accesses issued before the corresponding synchronization operation reads zero, i.e., it does not wait for any more access completions than is required by the algorithm. However, with just two

distinct classes and two counters, a similar problem will be faced if another synchronization operation is issued before the operations pending before the previous one are complete. At this point, the processor that issues the synchronization operation can be stalled. If deemed appropriate, more classes and counters can be used to further defer stalling.

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 await the completion of some set of previously issued reads and writes. Since normal read and write requests, and their corresponding invalidation and acknowledgement messages are never blocked at any time, they are guaranteed to complete. Hence a blocked processor will always unblock and termination is guaranteed.

### 4.4. A qualitative analysis

In this section, we show how the above implementation can yield higher performance than a straightforward implementation of Definition 1 (and sequential consistency), but still not as much as is made possible by Definition 2 with DRF. Consider the example illustrated by Figure 3. It shows an execution with two processors,

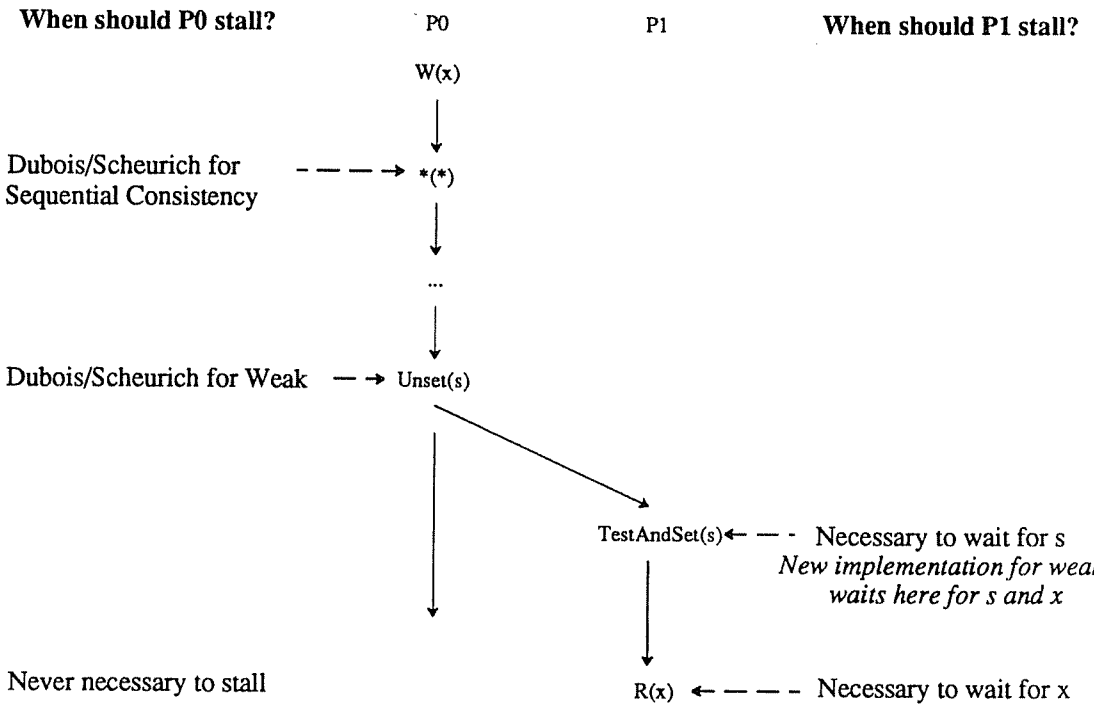


Figure 4. A qualitative comparison of the new implementation.

$P_0$ and $P_1$, sharing a data location x, and synchronizing on location s. Assume $P_0$ writes x, does other work and then *Unsets* s, after which $P_1$ *TestAndSets* s, does other work and then reads x.

Where and for how long must $P_0$ stall? $P_1$?

- *A straightforward implementation of sequential consistency.* $P_0$ and $P_1$ must stall at each access until its previously issued access completes [ScD87].
- *A straightforward implementation of Definition 1.* $P_0$ must stall at Unset(s) to complete previously issued accesses, including to x, before executing Unset(s). $P_1$ must stall at TestAndSet(s) until $P_0$'s Unset(s) is complete.
- *The implementation of Section 4.3.* $P_0$ may stall at some point after Unset(s) for implementation-dependent reasons (e.g., the cache block containing s needs to be replaced). $P_1$ must stall at TestAndSet(s) until $P_0$ completes accesses issued before and including Unset(s). (The algorithm of Section 4.1 never requires $P_0$ to stall.)
- *Definition 2 with DRF.* Lemma 1 of Appendix A shows it is necessary and sufficient for the TestAndSet(s) to wait until it can return the unset value and for the read of x to wait for the value from the write of x. Thus, $P_0$ need never stall, and $P_1$ could wait for s and x at the TestAndSet(s) and read(x), respectively.

While not achieving the full performance possible, the implementation of Section 4.3 yields higher performance than that implied by Definition 1 by allowing $P_0$ to immediately do other work after performing Unset(s), and never performing any worse than any implementation based on Definition 1.

## 5. An aggressive algorithm for sequential consistency

In the last section we saw how an implementation based on Definition 1 for weak ordering could be modified to yield new implementations with better performance potential. The new implementation of Section 4.3 also inspires a more aggressive implementation of sequential consistency for cache-based systems with general interconnects.

The conditions for sequential consistency in [ScD87, Sch89] require a processor to wait for its write operation to be visible to all other processors before it proceeds. Applying the same reasoning as for weak ordering, a processor can be allowed to proceed with an incomplete write, if it can be ensured that while the write is pending, the effect of its subsequent accesses is not visible to any other processor in the system. In effect, variables read and written by a processor while its previous write is pending, are treated like synchronization variables in the implementation of Section 4.3. In addition, to avoid rollback and deadlock, it is also required that while a write is pending, all subsequent accesses obtain the most recently written copy of the line. Furthermore, as in the case of

weak ordering, a processor is allowed to proceed on a write only after it has performed the write on some (probably its own) copy of the line. The complete algorithm, an implementation and a qualitative analysis follow.

### 5.1. Algorithm

A system is sequentially consistent if it obeys the following conditions. (Accesses below refer only to those on shared variables.)

1. Accesses are issued in program order.
2. All processors observe writes to a given location in the same order.
3. An access cannot be issued by a processor if

   (a) a previously issued read is not globally performed or

   (b) a previously issued write has not modified some copy of the line it accessed.
4. For a line on which a write $W$ issued by processor $P_i$ is globally performed while writes of $P_i$ before $W$ are incomplete,

   (a) a subsequent write by any processor can be globally performed only after all writes issued by $P_i$ before $W$ are globally performed.

   (b) a read by any other processor can return the value written by $W$ only after all writes issued by $P_i$ before $W$ are globally performed.
5. For a line on which a read $R$ issued by processor $P_i$ is globally performed while writes of $P_i$ before $R$ are incomplete, a subsequent write by any processor can be globally performed only after all writes issued by $P_i$ before $R$ are globally performed.
6. A read issued by a processor while some of its previous writes are incomplete, should return the value last written on any copy of the accessed line, where *last* is defined by the order ensured by condition 2.

*Proof of correctness*

The proof is based on a general technique we have developed for proving correctness of algorithms that ensure sequential consistency. This technique assigns a unique hypothetical timestamp to each access in an execution on a system that obeys the given algorithm. The timestamps are assigned so that an ordering of accesses in increasing order of their timestamps is consistent with the result of the execution and the program order. This implies sequential consistency. The details for this technique are given in Appendix D. In fact, the proofs for weak ordering described in Appendices A-C could as easily have been done by this method. The proof of correctness for the above algorithm using this general technique is provided in Appendix E. An informal intuition for the proof can be obtained by noting that conditions 4 and 5 ensure that while a processor has an incomplete write, the

effect of subsequent accesses is not seen by other processors. Hence to other processors, it was as if these accesses actually occurred after the pending writes. The timestamps are assigned in a manner that reflects this fact. In addition, it also needs to be proved that the above algorithm does not lead to deadlock. With the above conditions, deadlock can possibly occur only if a write is stalled indefinitely due to some cyclic dependencies between reads and writes of various processors as suggested by conditions 4 and 5. However, conditions 3b and 6 ensure that this is not possible, as described by the detailed proof in Appendix E.

### 5.2. Implementation

This section describes a possible implementation of the above algorithm on the system described in Section 4.2. We first give an implementation that allows only one write miss to be outstanding. Condition 1 is ensured directly by requiring a processor to issue accesses in program order. Condition 2 is satisfied due to the cache coherence protocol. For condition 3a, a processor is made to stall on a read until the requested line is procured. Recall that a read is globally performed only if the write whose value it returns is globally performed. For the case where a read hits in the processor cache, clearly condition 3a is satisfied. For the case where the read misses, ensuring the remaining conditions discussed below make it sufficient to wait until the requested line is procured by the processor. For condition 3b, a processor is made to wait on a write until it obtains the line in RW state in its cache and writes this copy of the line.

Conditions 4 and 5 are implemented in a manner analogous to the implementation of weak ordering in Section 4.3. A one-bit counter is associated with every processor and a reserve bit is associated with every cache line. The counter is set when a line originally in valid state is returned in response to a write request. It is reset on the receipt of an acknowledgement from memory indicating that the previous write to a line in valid state is globally performed. Similar to synchronization accesses in weak ordering, if a read or a write to a cache line is globally performed while the counter is set (indicating a previous incomplete write), the reserve bit of the line is set. In addition, when a write returns a line which was originally valid, its reserve bit is also set. All reserve bits are unset as soon as the counter is reset. Note that as for the weak ordering case, a reserved line that was written with the counter set is the only valid copy of the line in the system and hence the next request to this line is routed to the processor holding it reserved. Similarly, a write to a reserved line that was read with the counter set will result in an invalidate reaching the processor with the reserved line. These read, write and invalidation requests are not serviced as long as the line is reserved which is until the counter is reset thereby indicating the completion of the previous write. This ensures conditions 4a, 4b and 5. Again analogous to weak ordering, a processor may have to stall if it ever needs to flush a reserved line. To ensure condition 6, while the counter is set, only accesses to lines in RW state are considered hits. All the others are sent to the directory and the processor stalls until they are serviced.

The above implementation assumes only one incomplete write per processor at any given time. Hence, a processor is not allowed to issue a write miss until the counter is reset. It is not possible to use an n-valued counter to allow multiple outstanding writes as in the case of weak ordering. As for weak ordering, using such a scheme would imply that an access by another processor may have to be stalled for more writes to complete than required by conditions 4 and 5. Though this condition was important for the performance of the implementation of weak ordering, it did not compromise its correctness. However, for the case of sequential consistency, this situation can lead to deadlock (it violates a condition assumed in the proof of Appendix E). To handle multiple outstanding writes, extra hardware is required. One alternative is to keep track of exactly which acknowledgements were outstanding when any given access (say A) to a location (say x) is made. As the acks return, this information is updated. Now a request for x from another processor is serviced if and only if all outstanding acks when A was issued have returned. This is exactly what the algorithm requires. Such a scheme can be implemented to allow a limited number of outstanding writes.

### 5.3. A qualitative analysis

The implementation described in the previous subsection does not perform any worse than any based on the conditions given in [ScD87, Sch89]. However, it potentially performs better because of the following reasons.

(1) After issuing a write to a valid line, a processor is required to wait only until it receives its copy of the line. While invalidations corresponding to this write are being received and acknowledged by the other processors, all subsequent accesses of this processor that hit in the cache are immediately serviced. As opposed to this, the conditions given in [ScD87, Sch89] allow only accesses to local variables to be serviced while an ack is outstanding. Hence, particularly for systems built using off-the-shelf processors which do not have separate instructions to distinguish between local and shared accesses, the ability to process *all* hits could contribute to a reasonable speed-up by overlapping a large part of the latency of a shared write with the local (and shared) hits.

(2) As discussed in the implementation, with additional hardware, a processor can also be allowed to proceed with multiple outstanding write misses. This contributes to a further potential speed-up. Although the algorithm itself does not pose any limitations, for practical reasons, the number of such multiple outstanding requests, and hence the speed-up, may be limited.

Furthermore, the hardware requirements for the new implementations for weak ordering and sequential consistency are almost similar - both require counters and reserve bits, and the ability for a processor cache to block requests for reserved lines while the counter is positive. Thus our algorithms make it possible to implement both sequential consistency and weak ordering in the same system without much additional overhead. This is also true in the case of implementations based on the conditions given by Dubois et al. The sequentially consistent mode will give software more flexibility and can be used for purposes of debugging while the weakly ordered mode

gives higher performance.

## 6. More applications of the new definition: an alternate approach

In this section, we investigate a new approach for implementing weak ordering w.r.t. DRF that is made possible by Definition 2, but violates Definition 1.

The new approach is essentially a dual of that used so far. The methods discussed in previous sections have required a processor issuing a synchronization operation to wait until either all of its previous operations have been observed by all processors, or until all of the operations of the previous processor that accessed the synchronization variable are observed by all other processors. Thus these methods have essentially consisted of conditions of the following type.

> At some point ($t_2$) in the execution, a processor ($P_2$) blocks until all operations of *some processor* ($P_1$) before some point ($t_1$) are performed with respect to *all processors*. ($t_2$ may be the same as $t_1$ and $P_2$ may be the same as $P_1$.)

An alternative approach can be based on imposing the following type of condition.

> At some point ($t_2$) in the execution, a processor ($P_2$) blocks until all operations of *all processors* before some point ($t_1$) are performed with respect to *some processor* ($P_1$). ($t_2$ may be the same as $t_1$ and in the algorithms that follow, $P_2$ is the same as $P_1$.)

The term "performed with respect to a processor" used above was first defined by Dubois, Scheurich and Briggs in [DSB86]. A read is performed with respect to a processor when no subsequently issued store by the processor can affect the value returned by the read. A write is performed with respect to a processor when a subsequently issued read will return the value of this or a later write. ("Later" here is defined by the order in which writes are performed with respect to the processor.)

In the next subsection, we explore the implications of this new approach on weak ordering. As with the algorithm of Section 4.1, this also motivates a new algorithm for sequential consistency which is discussed in Section 6.2. While the work presented in this section may not yet be practical, it serves to illustrate the generality and applicability of Definition 2, and provides a promising direction for future research.

### 6.1. Weak ordering

Using the new approach with Definition 2, the following conditions suffice to ensure weak ordering w.r.t. DRF.

1. All processors observe writes on a given location in the same order, and dependencies within a processor are preserved.

2. Synchronization operations to the same location are totally ordered, i.e., appear atomic.

3. An access to a global variable is not issued by a processor until all its previous synchronization operations (in program order) are issued.

4. A synchronization operation is issued only after all previous accesses in program order have been issued.

5. A processor blocks on its synchronization operation until *all* accesses that are issued by *all* processors before the issue of the last synchronization operation to the same location are performed with respect to itself.

The above algorithm differs from that given in Section 4.1 primarily in condition 5. For ease of implementation, we introduce the notion of *committing* an access. A read is said to commit when the value it returns is fixed, i.e., cannot be changed by any subsequent write. Thus a read commits when it is performed with respect to all the processors. A write is said to commit when its position in the order defined by condition 1 is fixed. There is no equivalent to the committing of a write in the terms defined by Dubois et al. In particular, a write committing is not the same as it being performed with respect to the processor that issued it. This is because for most systems, the conditions for the latter event are true at the time the processor issues the write. A write in a system with general inter-connects will usually commit much after it is issued. With this notion of committing, conditions 4 and 5 can be modified as follows.

4′. A synchronization operation is issued only after all previous accesses in program order have been committed.

5′. A processor blocks on its synchronization operation until *all* writes that are committed by *all* processors before the issue of the last synchronization operation to the same location are performed with respect to itself.

As before, either Lemma 1 or the technique of Appendix E can be used to prove the correctness of the above sets of conditions. It is interesting to note that the scheme given by Bisiani et al. in [BNR89] and discussed in Section 2.2 obeys the above conditions *and* those imposed by the algorithm of Section 4.1.

A possible implementation of the above algorithm for a system like that discussed in Section 4.2 requires each processor to maintain a counter which approximately keeps track of the number of its issued, but uncommitted accesses. This is achieved by incrementing the counter when an access is issued and decrementing it when the corresponding line is procured. A synchronization operation is issued only if the counter reads zero. This ensures condition 4′.

For condition 5′, consider a processor $P_i$ that is blocked on a synchronization operation. A committed read is performed with respect to all processors. Hence for condition 5′, we need only ensure that all writes committed before the last synchronization operation are performed with respect to $P_i$. With a cache coherency protocol like that described in Section 4.2, a write is committed only after it reaches memory (or the directory). So $P_i$ is required to send a message to all the memory modules from which it can possibly receive invalidations. These

modules then send acks to $P_i$ through all the paths which can carry invalidations for $P_i$. If it can be ensured that messages do not pass each other on any given path, the receipt of all these acks by $P_i$ would indicate that it has received all required invalidations. It can then proceed with its next access.

The above implementation requires extra messages to be transmitted on the network for every synchronization operation. As opposed to this, the implementation of Section 4.3 required acks for every write to a line in valid state. Thus it is possible (particularly with broadcast invalidates), that the above implementation may result in lower bandwidth requirements. However, the latency of synchronization operations appears to be much larger than that in Section 4.3.

An alternative way to ensure condition 5′ is to have a central controller in the system which (1) assigns monotonically increasing timestamps to all invalidation messages, and to all accesses to synchronization variables, and (2) routes all invalidations to processors so that they reach their destinations in the order of increasing timestamps. Furthermore, a write to a valid line is not considered committed, and the line is not returned to the requesting processor until the central controller dispatches its invalidations.

Every copy of a synchronization variable also contains the timestamp of the last access to this variable. Every processor maintains a register *lastI* which keeps track of the timestamp of the last invalidation message it received. If on a synchronization access, the timestamp of the synchronization variable procured is less than lastI, the processor can proceed. If this is not the case, then the processor sends a message to the central controller. An ack from the central controller ensures that the processor has received all required invalidations and it can proceed.

Timestamps for synchronization accesses do not necessarily have to be obtained from the central controller. A register *lastW* is associated with every processor. This stores the timestamp of the invalidations from the last committed write issued by the processor for a valid line. Then after a synchronization access completes, it is sufficient to update the timestamp of the variable accessed to the value in lastW.

### 6.2. Sequential consistency

A new set of conditions using the above approach for implementing sequential consistency is as follows.

1. Accesses are issued in program order.
2. All processors observe writes to a given location in the same order.
3. After a write is issued by processor $P_i$, no further access can be issued by $P_i$ until the write is committed.
4. A read by processor $P_i$ is not issued until *all* writes of *all* processors committed before the previous write of $P_i$ have been performed with respect to $P_i$.

5. After a read is issued by processor $P_i$, no further access can be issued by $P_i$ until all writes committed before the write whose value the read returned, are performed with respect to $P_i$.

Correctness of the above conditions can be proved using the technique of Appendix E. The timestamps required by the proof can be assigned as follows. For a write, the timestamp is assigned to reflect the time at which it is committed. For a read, the timestamp should reflect the greater of the times at which the write whose value it returns, and the last write issued by its processor are committed.

Note that Condition 5 bears a strong resemblance to the necessary condition for strong ordering given by Dubois et al. in [DSB86]. Intuitively this algorithm is a direct attempt to ensure that all writes to all locations appear to have occurred to every processor in the same order. As discussed in Section 2.1, Collier has proved that this is equivalent to write atomicity and hence ensures sequential consistency.

The central controller scheme outlined in the previous subsection for weak ordering can be extended to implement the above conditions. In this case, every location is associated with a timestamp reflecting the time at which the last write to it was committed, and every read is treated like a synchronization access in the implementation of weak ordering. On a read access, the processor uses the values in the registers lastI and lastW along with the timestamp of the accessed line to determine if it has received all the required invalidations. If not, as before it sends a message to the central controller.

## 7. Conclusions

Most programmers implicitly assume the model of sequential consistency. For shared memory multiprocessors, 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. We believe, however, that this definition is unnecessarily restrictive on hardware and does not clearly specify the programmer's view of a system. Intuitively, a weakly ordered system promises to be sequentially consistent only if software obeys a certain set of constraints, which we call the synchronization model. We define a system to be weakly ordered with respect to a synchronization model. We believe our definition reflects the intuition behind weak ordering, gives a clear model for the programmer and does not impose unnecessary directives for hardware.

We have formally specified a synchronization model, called Data-Race-Free (DRF), which does not restrict the synchronization mechanism, as long as it is visible to hardware. We believe that DRF is not too restrictive to the programmer either, and is quite compatible with the way most programs are currently written. The new definition of weak ordering with DRF allows implementations that violate the old definition and have a higher performance potential than any others based on the old definition. For example, unlike the old definition, the new

definition does not require a processor to ensure that all other processors have observed its previous accesses before issuing a synchronization operation. One such implementation has been described.

The insight gained from weak ordering has also been applied to give a new algorithm for sequential consistency. A new approach allowed by the new definition for implementing weak ordering w.r.t. DRF, and its extension to sequential consistency has been outlined. A promising direction for future research is an application of the new definition in a further exploration of alternative implementations. A quantitative performance analysis comparing implementations for the old and new definitions of weak ordering would provide useful insight.

Another interesting problem for future research is the construction of other synchronization models that may allow implementations with higher performance. These could be optimized for particular synchronization primitives that may be offered by specific systems, e.g., QOLB in the Wisconsin Multicube [GVW89].

## 7. Acknowledgements

We would like to thank William Collier, Garth Gibson, Richard Kessler, Viranjit Madan, Bart Miller and Robert Netzer for their useful comments on earlier drafts of the paper.

## References

[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.

[AST67] D. W. Anderson, F. J. Sparacio and R. M. Tomasulo, The IBM System/360 Model 91: Machine Philosophy and Instruction-Handling, *IBM Journal*, January 1967, 8-24.

[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.

[Arc87] J. Archibald, The Cache Coherence Problem in Shared-Memory Multiprocessors, Ph.D. Thesis, University of Washington, March 1987.

[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.

[Col84a] W. W. Collier, Architectures for Systems of Parallel Processes, Technical Report Tech. Rep. 00.3253, IBM Corp., Poughkeepsie, N.Y., 27 January 1984.

[Col84b] W. W. Collier, Write Atomicity in Distributed Systems, Technical Report Tech. Rep. 00.3304, IBM Corp., Poughkeepsie, N.Y., 19 October 1984.

[Col85] W. W. Collier, Reasoning about Parallel Architectures, Technical Report, IBM Corp., Poughkeepsie, N.Y., 1985.

[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.

[DuS89] M. Dubois and C. Scheurich, Private Communication, November 1989.

[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.

[GGK83] A. Gottlieb, R. Grishman, C. P. Kruskal, K. P. McAuliffe, L. Rudolph and M. Snir, The NYU Ultracomputer--Designing an MIMD Shared Memory Parallel Computer, *IEEE Trans. on Computers C-32*, 2 (February 1983), 175-189.

[Jef85] D. R. Jefferson, Virtual Time, *ACM Trans. on Programming Languages and Systems 7*, 3 (July 1985), 404-425.

[Kni86] T. Knight, An Architecture for Mostly Functional Languages, *Proc. ACM Conference on LISP and Functional Programming*, 1986, 105-112.

[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.

[Lam86a] L. Lamport, The Mutual Exclusion Problem, Parts I and II , *Journal of the Association of Computing Machinery 33*, 2 (April 1986), 313-348.

[Lam86b] L. Lamport, On Interprocess Communication, Parts I and II , *Distributed Computing* , January 1986, 78-101.

[Lam87] L. Lamport, A Fast Mutual Exclusion Algorithm, *ACM Transactions on Computer Systems 5*, 1 (February 1987), 1-11.

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

[Nic89] K. E. Nicholas, Levels of Ordering and Consistency in Shared-Memory Multiprocessors, Master's Thesis, Department of Electrical and Computer Engineering, Brigham Young University, April 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.

[SmP88] J. E. Smith and A. R. Pleszkun, Implementing Precise Interrupts in Pipelined Processors, *IEEE Trans. on Computers C-37*, 5 (May 1988), 562-573.

[TiK88] P. Tinker and M. Katz, Parallel Execution of Sequential Scheme with ParaTran, *Proc. ACM Conference on LISP and Functional Programming*, July 1988, 28-39.

**Appendix A: A necessary and sufficient condition for weak ordering w.r.t. DRF.**

**Lemma 1:** A system is weakly ordered w.r.t. DRF by Definition 2 if only if for *any* execution of a program that obeys DRF, there exists a happens-before relation $\xrightarrow{hb}$, defined for the program such that a read always returns the value written by the last write[5] on the same variable, ordered 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 w.r.t. DRF but does not obey the above condition. Then for any execution E of a program that obeys DRF, there must be a total ordering T of all its accesses which produces the same result as E and which is consistent with the program order of all 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 $E'$. This is the irreflexive, transitive closure of $\xrightarrow{po}$ and $\xrightarrow{so}$ as defined by T. 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 DRF) before it by $\xrightarrow{hb}$. Since in the general case, 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 return the value of the last write ordered before it by $\xrightarrow{hb}$. This contradicts our hypothesis. □

5. A read is a normal read or a synchronization operation that returns a value. A write is a normal write or a synchronization operation that modifies memory. 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.

*Proof of sufficiency*

We now prove that a system which obeys the given condition is weakly ordered w.r.t. DRF. From the given condition, there exists a happens-before relation $\xrightarrow{hb}$ corresponding to any execution $E$ of a program that obeys DRF, such that every read 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 $E'$ (this is further discussed in the next paragraph). 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. □

The lemma as stated does not explicitly account for the initial and final state of memory, and I/O operations. The initial state of memory can be incorporated by adding hypothetical initialization writes before the start of the execution, while the final state is incorporated by introducing hypothetical final reads after the termination of the execution. For I/O operations, the lemma does not impose any constraints on the order in which they may be executed. Though this does not violate sequential consistency of *memory* operations, practical considerations may necessitate that all related I/O operations be executed in a manner consistent with the happens-before relation identified for the execution. We concentrate on memory operations, and in the proofs that follow ignore the condition for I/O.

**Appendix B: Proof that a system that obeys Definition 1 is also weakly ordered w.r.t. DRF by Definition 2.**

The proposition is proved by showing that a system that obeys Definition 1 also satisfies the condition of Lemma 1 in Appendix A. The proof makes the following assumptions for an implementation with Definition 1:

(A1) Intra-processor dependencies are always satisfied.

(A2) Writes to a *given* location by any given processor are observed by all the other processors in the same order.

(A3) Accesses to synchronization variables are not just strongly ordered as claimed by condition 1 of the algorithm, but also totally ordered.

The proof proceeds by contradiction. Suppose for some program that obeys DRF, there exists an execution E on a system that obeys Definition 1, that violates the condition of Lemma 1. Defining $\xrightarrow{so}$ as the order of completion of synchronization operations in E and $\xrightarrow{po}$ as the program order relation for the accesses in E, a happens-before relation can be constructed. This is possible because Condition 2 of Definition 1 ensures that $\xrightarrow{so}$ is consistent with $\xrightarrow{po}$. From our hypothesis there exists some read $R$ in E that returns the value of a write $W'$ that

is not the last write ordered before $R$ by $\xrightarrow{hb}$. Let $W$ be this last write. DRF ensures that $W$ is unique. Since A3 ensures that accesses to synchronization variables are totally ordered, and since the above $\xrightarrow{hb}$ is consistent with their order of execution, we need only worry about accesses to non-synchronization variables.

DRF ensures that either $W' \xrightarrow{hb} W \xrightarrow{hb} R$ or $W \xrightarrow{hb} R \xrightarrow{hb} W'$. Consider the former case. If $W' \xrightarrow{po} W$, then by A1 and A2, every processor observes $W'$ before $W$. If $W'$ and $W$ are ordered by intervening synchronization operations, $S_0, S_1, S_2, \cdots, S_n$, then condition 2 ensures that $W'$ is globally performed before $S_0$ is issued and condition 3 ensures that $W$ is not issued until $S_n$ is globally performed. Since $\xrightarrow{hb}$ is defined in terms of the order of execution of synchronization operations, it follows from condition 1 that $W'$ is observed before $W$ by every processor. Using a similar argument, we can show that the processor that issues $R$ always observes $W$. It follows that $R$ cannot return the value written by $W'$.

For the case where $W \xrightarrow{hb} R \xrightarrow{hb} W'$, an argument similar to the above proves that $W$ is always observed by $R$ and $R$ never observes $W'$. Hence $R$ can never return the value written by $W'$. This contradicts our initial hypothesis. □

**Appendix C: Proof of correctness of aggressive algorithm for weak ordering with DRF (Section 4.1).**

The proof proceeds in a manner similar to that in Appendix B. We prove by contradiction that a system that obeys the algorithm of Section 4.1 also obeys the necessary and sufficient condition of Lemma 1 in Appendix A. Suppose for some program that obeys DRF, there exists an execution E on a system that obeys the algorithm of Section 4.1, but violates the condition of Lemma 1. Defining $\xrightarrow{so}$ as the order of completion of synchronization operations in E and $\xrightarrow{po}$ as the program order relation for the accesses in E, a happens-before relation can be constructed. This is possible because condition 3 of the algorithm ensures that $\xrightarrow{so}$ is consistent with $\xrightarrow{po}$. From our hypothesis there exists some read $R$ in E that returns the value of a write $W'$ that is not the last write ordered before $R$ by $\xrightarrow{hb}$. Let $W$ be this last write. DRF ensures that $W$ is unique. Condition 2 of the algorithm ensures that accesses to synchronization variables are totally ordered, and since the above $\xrightarrow{hb}$ is consistent with their order of execution, we need only worry about accesses to non-synchronization variables.

DRF ensures that either $W' \xrightarrow{hb} W \xrightarrow{hb} R$ or $W \xrightarrow{hb} R \xrightarrow{hb} W'$. Consider the former case. If $W' \xrightarrow{po} W$, then by condition 1 of the algorithm, every processor observes $W'$ before $W$. If $W'$ and $W$ are ordered by intervening synchronization operations, $S_0, S_1, S_2, \cdots, S_n$, then condition 4 ensures $S_1$ is not serviced until $W'$ is globally performed. Condition 3 ensures that $W$ is not issued until $S_n$ is performed. Since $\xrightarrow{hb}$ is defined in terms of the order of execution of synchronization operations, it follows that $W'$ is observed before $W$ by every processor. Since $W \xrightarrow{hb} R$, using a similar argument, we can show that the processor that issues $R$ always observes $W$. It follows that $R$ cannot return the value written by $W'$.

For the case where $W \xrightarrow{hb} R \xrightarrow{hb} W'$, an argument similar to the one above proves that $W$ is always observed by $R$ and $R$ never observes $W'$. Hence $R$ can never return the value written by $W'$. This contradicts our initial hypothesis. □

To incorporate the relaxation on condition 3 mentioned in Section 4.1, construct the $\xrightarrow{so}$ relation based on the order in which processors successfully perform synchronization operations on their copy of the corresponding synchronization variable. The rest of the proof follows.

**Appendix D : General methodology for proofs for sequential consistency**

We have developed a general methodology for proving the correctness of algorithms for sequential consistency. We believe this methodology to be quite general and useful, and outline it below.

*Step 1*

Assign a unique timestamp to all accesses. The timestamp will most generally be an n-tuple $<t_1, t_2, \cdots, t_n>$ where n ≥ 1. A total ordering can be established on the timestamps depending on the values of the various fields in the n-tuple, $t_1$ being regarded as the most significant, then $t_2$, etc.

*Step 2*

Claim that the ordering of accesses according to the increasing order of their timestamps is an equivalent serial order for any execution.

*Step 3*

Prove the above order of accesses is consistent with program order

*Step 4*

Prove the claim of step 2 to be true by showing that in every execution, a read always returns the value written by the last write (to the same variable) before the read, in timestamp order. This can be done by proving the following four cases:

- A read sees the effect of all writes to the same variable with timestamp less than its own.
- A read does not see the effect of any of the writes to the same variable with timestamp greater than its own.
- A write sees the effect of all writes to the same variable with timestamp less than its own.
- A write does not see the effect of any of the writes to the same variable with timestamp greater than its own.

Above, a read is said to see the effect of a write if it will return a value written by this or subsequent writes. A write $W_1$, is said to see the effect of a write $W_2$, if no subsequent read will return the value written by $W_2$.

*Step 5*

Prove that the algorithm never leads to deadlock.

Since private data does not pose any problem for sequential consistency, the proofs need only concern data shared by two or more processors.

**Appendix E : Proof of correctness of the algorithm for sequential consistency in Section 5**

*Step 1* - Assigning timestamps

Accesses which are globally performed while the issuing processor has previous writes outstanding, or at the same time as a previous write, will be referred to as *special* accesses. Accesses which are not special will be referred to as *normal* accesses. A state in which a processor can issue accesses while previously issued writes are incomplete will be referred to as a pending write state.

Consider a hypothetical counter which is incremented whenever an access is globally performed. A unique timestamp which is a 3-tuple < counter, index, pid > is assigned to each access in an execution as follows:

For a normal access,

counter = the counter value at which the access was globally performed

index = 0

pid = unique identification for the processor that issued the access.

For a special access,

counter = the counter value at which the last of the incomplete writes,
issued before the special access, was globally performed

index = position of this access within its process in program order

pid = unique identification for the processor that issued the access.

Note that the write whose counter value a special access is assigned, is a normal write. The special access is said to be *associated* with this normal write. The timestamp of access $a_1$ will be referred to as TS($a_1$). The counter value in the timestamp of access $a_1$ will be referred to as counter($a_1$).

The pid field in the timestamps serves to disambiguate accesses that have the same counter and index value, thereby resulting in every access having a unique timestamp. (Accesses of the same processor cannot have the same counter and index values.) For now we will assume that no two normal accesses can be globally performed at the same counter value. Later, we will show that even if they are, their relative ordering does not matter, and so they can be ordered on the basis of their pid field.

*Step 2 - Claim*

We claim that ordering the accesses in the order of increasing timestamps is an equivalent sequential order for any execution.

*Step 3 - Proof that accesses in the above order are consistent with program order.*

The relative ordering between the following pairs of accesses issued by a processor needs to be verified.

(1) Two normal accesses - These are globally performed in the order in which they are issued, i.e, program order (condition 1, 3a). Hence the timestamp ordering is consistent with program order.

(2) Two special accesses associated with the same write - These have the same counter value. The index value is consistent with program order. Hence timestamp order is consistent with program order.

(3) A special access $A$ and the normal write $W$ it is associated with - Again the counter value is the same. Index of $W = 0$, while that of $A$ is positive. Clearly $A$ has to be ordered after $W$ by the program. Hence the timestamp ordering for $A$ and $W$ is consistent with program order.

(4) A special access $A$, and a normal write $W$, with which it is not associated - Let $W'$ be the normal write with which $A$ is associated. If $W$ was issued before $W'$, then it should have been globally performed before $W'$ (otherwise, $A$ would be associated with $W$). Hence $W$ is ordered before $A$ by timestamps and the program. If $W$ was issued after both $W'$ and $A$, then it should have been globally performed after $W'$ since otherwise it would not be a normal write. In this case, $W$ is ordered after $A$ by timestamps and the program. If $W$ was issued between $W'$ and $A$, it has to be either a special write or the normal write associated with $A$ and hence this case is not possible.

(5) A special access $A$, and a normal read $R$ - Let $W$ be the normal write with which $A$ is associated. If $R$ was issued before $W$, then it was globally performed before $W$ (condition 3a). Hence $R$ is ordered before $A$ by timestamps and the program. If $R$ was issued after both $W$ and $A$, then it should have been globally performed after $W$ since otherwise it would not be a normal read. In this case, $R$ is ordered after $A$ by timestamps and the program. If $R$ was issued between $W$ and $A$, it was globally performed before $A$ and hence is a special read which is a contradiction.

(6) Two special accesses $A_1$ and $A_2$ not associated with the same normal write - An argument similar to the previous cases can be applied to this case as well to prove that timestamp ordering of $A_1$ and $A_2$ is consistent with program order.

*Step 4 - Proof that for every execution, a read always returns the value written by the last write to the same variable before the read in timestamp order.*

The following results follow directly from the definition of timestamps, globally performed, and the algorithm:

(a) Condition 2 ensures that an access observes the effect of a write if and only if it is globally performed after the write.

(b) The timestamp of a normal access reflects the counter value at which it was globally performed.

(c) A special access is globally performed at a counter value less than or equal to that in its timestamp.

(d) From results b and c, all accesses are globally performed at counter values less than or equal to those in their timestamps.

We now use the above results to prove the following four cases:

Case 1: A read $R$, observes the effect of a write $W$, where TS($R$) > TS($W$).

Results a, b and d ensure that if $R$ is a normal read, it will observe the effect of $W$. If $R$ is special, let it be globally performed at counter value $t_1$ < counter($R$). From condition 5, no write on $x$ can be globally performed between $t_1$ and counter($R$). If $W$ is globally performed before $t_1$, then $R$ observes the effect of $W$. If $W$ is globally performed after counter($R$), then TS($R$) < TS($W$), a contradiction.

Case 2: A read $R$, cannot observe the effect of a write $W$, where TS($R$) < TS($W$).

Results a, b and d ensure that if $W$ is a normal write, its effect cannot be observed by $R$. If $W$ is special, then condition 4b ensures that $R$ returns the value written by $W$ only if TS(R) > TS(W). This is a contradiction and hence $R$ does not observe the effect of $W$.

Case 3: A write $W_1$, observes the effect of a write $W_2$, where TS($W_1$) > TS($W_2$).

This can be proved by an argument similar to that for case 1, using condition 4a instead of condition 5.

Case 4: A write $W_1$, cannot observe the effect of a write $W_2$, where TS($W_1$) < TS($W_2$).

Again condition 4a can be used to show that this is true.

*Step 5* - Proof that the algorithm never causes deadlock.

An access issued by a processor $P_0$ can only be stalled due to a processor $P_1$ in pending write state i.e. previous writes of $P_1$ are not globally performed. A write to a line by processor $P_1$ is not allowed to be globally performed only if the line was written or read while a processor $P_2$ was in pending write state, and these pending writes have not been globally performed (conditions 4 and 5). It is possible that the pending writes of $P_2$ are held up for the same reason by the pending writes of some other processor, say $P_3$. We now prove that this chain of pending writes cannot lead to a cycle. Let $W_1 = write(P_1,x)$ be the write that is issued by $P_1$ that is blocked because of an access $A_2$ issued by $P_2$ on $x$ while its write $W_2 = write(P_2,y)$ was pending. Let $W_2$ be blocked by the access $A_3$ issued by $P_3$ on $y$ which was globally performed while $W_3 = write(P_3,z)$ was pending. $A_2$ can block $W_1$ only if it is globally performed before $W_1$ (conditions 4a and 5). Since $A_2$ was issued while $P_2$ had a pending write, conditions 2 and 6 ensure that $A_2$ was issued before $P_1$ got into a pending write state due to $W_1$. Thus $P_2$

got into a pending write state due to $W_2$ before $P_1$ did so due to $W_1$. Since $W_2$ can only be stalled by an access globally performed before it put $P_2$ into the pending write state (conditions 2, 3b and 6), $W_2$ cannot be stalled due to any accesses issued by $P_1$ after and including $W_1$. By the same argument, $W_3$ put $P_3$ into pending write state before $W_2$ did so to $P_2$ and before $W_1$ did so to $P_1$. Hence $W_3$ cannot be stalled by any accesses issued by $P_2$ and $P_1$ after and including $W_2$ and $W_1$ respectively. Extending this argument to all writes and all processors implies that there can never be a cycle in the above chain. Thus there is no deadlock.

*Incorporating multiple accesses globally performed at the same counter value.*

It remains to be shown that our claim holds even if more than one access is globally performed with the same counter value. We will only consider normal reads and writes since special accesses are assigned the same counter value as the normal write they are associated with.

By definition, two *normal* accesses can be globally performed at the same counter value only if they are issued by different processors. Since a read is considered to be globally performed only when the write whose value it returns is, we assume that a read and a write to the same location can also never be globally performed at the same counter value. Hence in general, two accesses which are globally performed at the same counter value but do not put their respective processors in the pending write state can be ordered arbitrarily with respect to each other.

Now consider two (normal) accesses that are globally performed at the same counter value where one of them is $R = read(P_1,x)$, and the other is $W = write(P_2,y)$ that puts $P_2$ in a pending write state. If there was a special access $a_1$ issued on $x$ by $P_2$ after $W$, then the order between $R$ and $a_1$, and hence $W$ is important. From the definition of a special access, $a_1$ is globally performed before or at the same time as $W$, and hence $R$. If $a_1$ is a write, then it should have been globally performed before $R$, and $R$ is required to return the value written by $a_1$ or a subsequent write. By condition 4a, no subsequent write can be globally performed before $R$. Hence $R$ is required to return the value written by $a_1$. But condition 4b does not allow this until $W$ is globally performed. Hence if $a_1$ is a write access, $R$ cannot be globally performed until after $W$. This is a contradiction. If $a_1$ is a read, then condition 5 ensures that no write can be globally performed after $a_1$ and before $W$. Hence $R$ returns the same value as $a_1$ and the relative ordering between $W$ and $R$ can be arbitrary. Using conditions 4a and 5, we can similarly show that for two normal accesses $W_1$ and $W_2$ which are globally performed at the same counter value, and only one (say $W_1$) puts its processor in the pending write state, there cannot be a special access associated with $W_1$ that accesses the same location as $W_2$. Hence $W_1$ and $W_2$ can also be arbitrarily ordered.

This leaves the case of two normal writes, $W_1 = write(P_1,x)$ and $W_2 = write(P_2,y)$ that are globally performed at the same counter value and both put their processors in pending write state. From the above arguments, none of the special accesses associated with $W_1$ can access a location accessed by $W_2$ and vice versa. Similarly,