Computer Sciences Technical Report #1051, September 1991, Revised September 1992

A Unified Formalization of Four Shared-Memory Models

Sarita V. Adve
Mark D. Hill

Department of Computer Sciences
University of Wisconsin
Madison, Wisconsin 53706

ABSTRACT

This paper presents a shared-memory model, data-race-free-1, that unifies four earlier models: weak ordering, release consistency (with sequentially consistent special operations), the VAX memory model, and data-race-free-0. The most intuitive and commonly assumed shared-memory model, sequential consistency, limits performance. The models of weak ordering, release consistency, the VAX, and data-race-free-0 are based on the common intuition that if programs synchronize explicitly and correctly, then sequential consistency can be guaranteed with high performance. However, each model formalizes this intuition differently and has different advantages and disadvantages with respect to the other models.

Data-race-free-1 unifies the models of weak ordering, release consistency, the VAX, and data-race-free-0 by formalizing the above intuition in a manner that retains the advantages of each of the four models. A multiprocessor is data-race-free-1 if it guarantees sequential consistency to data-race-free programs. Data-race-free-1 unifies the four models by providing a programmer's interface similar to the four models, and by allowing all implementations allowed by the four models. Additionally, data-race-free-1 expresses the programmer's interface more explicitly and formally than weak ordering and the VAX, and allows an implementation not allowed by weak ordering, release consistency, or data-race-free-0.

The new implementation proposal for data-race-free-1 differs from earlier implementations mainly by permitting the execution of all synchronization operations of a processor even while previous data operations of the processor are in progress. To ensure sequential consistency, two synchronizing processors exchange information to delay later operations of the second processor that conflict with an incomplete data operation of the first processor.

Index Terms: data-race-free-0, data-race-free-1, memory model, release consistency, sequential consistency, shared-memory multiprocessor, weak ordering.

This work was supported in part by a National Science Foundation Presidential Young Investigator Award (MIPS-8957278) with matching funds from A.T. & T. Bell Laboratories, Cray Research Foundation, and Digital Equipment Corporation. Sarita Adve is also supported by an IBM graduate fellowship.

1. Introduction

A memory model for a shared-memory multiprocessor system is a formal specification of how memory operations in a program will appear to execute to the programmer. In particular, a memory model specifies the values that may be returned by read operations executed on a shared-memory system. This paper presents a new memory model, data-race-free-1, that unifies four earlier models.[1] Although the four models are very similar, each model has different advantages and disadvantages for programmers and system designers. Data-race-free-1 unifies the four models by retaining all the advantages of the four models.

Most uniprocessors provide a simple memory model that ensures that memory operations will appear to execute one at a time, in the order specified by the program (program order). Thus, a read returns the value from the last write (in program order) to the same location. To improve performance, however, uniprocessors often allow memory operations to overlap other memory operations and to be issued and executed out of program order. Uniprocessors use interlock logic to maintain the programmer's model of memory (that memory operations appear to execute one at a time, in program order). This model of uniprocessor memory, therefore, has the advantage of simplicity and yet allows for high performance optimizations.

The most commonly (and often implicitly) assumed memory model for shared-memory multiprocessor systems is *sequential consistency*, formalized by Lamport [21] as follows.

> **Definition 1.1:** [A multiprocessor system is *sequentially consistent* if and only 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.

In other words, a sequentially consistent multiprocessor appears like a multiprogrammed uniprocessor [24].

Although sequential consistency retains the simplicity of the uniprocessor memory model, it limits performance by preventing the use of several optimizations. Figure 1 shows that in multiprocessor systems, both with and without caches, common uniprocessor hardware optimizations, such as write buffers, overlapped memory operations, out-of-order memory operations, and lockup-free caches [20], can violate sequential consistency. These optimizations significantly improve performance and will become increasingly important in the future as processor cycle times decrease and memory latencies increase [13]. Gharachorloo et al. have described

1. An earlier version of this work appears in the Proceedings of the 17th Annual International Symposium on Computer Architecture, June 1990 [1]. The data-race-free-1 memory model developed in this paper extends the data-race-free-0 model of [1] by distinguishing unpaired synchronization operations from paired release and acquire synchronization operations. The definition of data-race-free-1 in Section 2 uses the notions of how different operations are distinguished, when the distinction is correct, the synchronization-order-1 and happens-before-1 relations, and data races. These notions are extensions of similar concepts developed for data-race-free-0. Also, in parallel with our work on this paper, we published a technique for detecting data races on a data-race-free-1 system [2]. Consequently, [2] reviews data races and the data-race-free-1 memory model, and contains the definitions (in slightly different form) of Section 2. This material is used in Section 2 with the permission of the ACM.

mechanisms that allow these optimizations to be used with the sequential consistency model, but the mechanisms require hardware support for prefetching and rollback [12].

Initially X = Y = 0

P_1	P_2
X = 1	Y = 1
r1 = Y	r2 = X

Result: r1 = r2 = 0

Figure 1. A violation of sequential consistency.[2]

X and Y are shared variables and r1 and r2 are local registers. The execution depicted above violates sequential consistency since no total order of memory operations consistent with program order lets both P_1 and P_2 return 0 for their reads on Y and X. Note that neither processor has data dependencies among its instructions; therefore, simple interlock logic will not preclude either processor from issuing its second instruction before the first.

Shared-bus systems without caches - The execution is possible if processors issue memory operations out of order or allow reads to pass writes in write buffers.

Systems with general interconnection networks without caches - The execution is possible even if processors issue memory operations in program order, if the operations reach memory modules in a different order [21].

Shared-bus systems with caches - Even with a cache coherence protocol [6], the execution is possible if processors issue memory operations out-of-order or allow reads to pass writes in write buffers.

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

Alternate memory models have been proposed to improve the performance of shared-memory systems. To be useful, the new models should satisfy the following properties: (1) the model should be simple for programmers to use, and (2) the model should allow high performance. The central assumption of this work is that most programmers prefer to reason with the sequential consistency model since it is a natural extension of the well-understood uniprocessor model. Therefore, one way in which a memory model can satisfy the first property is to appear sequentially consistent to most programs and to formally characterize this group of programs. A memory model can satisfy the second property by allowing all high performance optimizations that guarantee sequential consistency for this group of programs.

One group of programs for which it is possible to guarantee sequential consistency and still use many optimizations is programs that explicitly distinguish between synchronization memory operations (operations used to order other operations) and data memory operations (operations used to read and write data). This dichotomy

2. Figure 1 is a modified version of Figure 1 in [1] and is presented with the permission of the IEEE.

between memory operations is the motivation behind the four models of weak ordering [9], release consistency with sequentially consistent special operations (henceforth called *release consistency*) [11], the VAX [8], and data-race-free-0 (originally called weak ordering with respect to data-race-free-0) [1].

Although the four memory models are very similar, small differences in their formalization lead to differences in the way they satisfy the above two properties. Weak ordering [9] and release consistency [11] restrict hardware to actually execute specific memory operations in program order. For programmers, the authors of weak ordering later stated that mutual exclusion should be ensured for each access to a shared variable by using constructs such as critical sections, which are implemented with hardware-recognizable synchronization operations [10, 26]. The authors of release consistency formalize a group of programs called *properly labeled programs*, for which release consistency ensures sequential consistency. A properly labeled program distinguishes its memory operations depending on their use. For example, it distinguishes synchronization operations from ordinary data operations. The VAX and data-race-free-0 models differ from weak ordering and release consistency by avoiding explicit restrictions on the actual order of execution of specific memory operations. In the VAX architecture handbook [8], the data sharing and synchronization section states the following. ‘‘Accesses to explicitly shared data that may be written must be synchronized. Before accessing shared writable data, the programmer must acquire control of the data structure. Seven instructions are provided to permit interlocked access to a control variable.’’ Data-race-free-0 [1] states that sequential consistency will be provided to data-race-free programs. A data-race-free program (discussed formally in Sections 2 and 3) distinguishes between synchronization operations and data operations and ensures that conflicting data operations do not race (i.e., cannot execute concurrently). For programs that contain data races, data-race-free-0 does not guarantee the behavior of the hardware.

The different formalizations of the four models result in some models satisfying the simplicity or the high-performance property better than other models; however, no model satisfies both properties better than all other models. For example, the VAX imposes the least restrictions on hardware, but its specification is less explicit and formal than the other models. Consider the statement, ‘‘before accessing shared writable data, the programmer must acquire control of the data structure.’’ Does this allow concurrent readers? Further, how will hardware behave if programs satisfy the specified conditions? Although it may be possible to answer these questions from the VAX handbook, a more explicit and formal interface would allow a straightforward and unambiguous resolution of such questions. Release consistency, on the other hand, provides a formal and explicit interface. However, as Section 4 will show, the hardware requirements of release consistency are more restrictive than necessary.

This paper defines a new model, data-race-free-1, which unifies the weak ordering, release consistency, VAX, and data-race-free-0 models in a manner that retains the advantages of each of the models for both the programmer and the hardware designer. The following summarizes how data-race-free-1 unifies the four models and how it overcomes specific disadvantages of specific models.

For a programmer, data-race-free-1 unifies the four models by explicitly addressing two questions: (a) when is a program correctly synchronized? and (b) how does hardware behave for correctly synchronized programs? Data-race-free-1 answers these questions formally, but the intuition behind the answers is simple: (a) a program is correctly synchronized if none of its sequentially consistent executions have a data race (i.e., conflicting data operations do not execute concurrently), and (b) for programs that are correctly synchronized, the hardware behaves as if it were sequentially consistent. This viewpoint is practically the same as that provided by release consistency and data-race-free-0. However, it is more explicit and formal than weak ordering and the VAX (e.g., it allows concurrent readers because they do not form a data race).

For a hardware designer, data-race-free-1 unifies the four models because (as will be shown in Section 4) implementing any of the models is sufficient to implement data-race-free-1. Furthermore, data-race-free-1 is less restrictive than either weak ordering, release consistency, or data-race-free-0 for hardware designers since there exists an implementation of data-race-free-1 that is not allowed by weak ordering, release consistency, or data-race-free-0. The new implementation (described in Section 4) differs from implementations of weak ordering and release consistency by allowing synchronization operations to execute even while previous data operations of the synchronizing processors are incomplete. To achieve sequential consistency, processors exchange information at the time of synchronization that ensures that a later operation that may conflict with an incomplete data operation is delayed until the data operation completes. The new implementation differs from implementations of data-race-free-0 by distinguishing between different types of synchronization operations.

The rest of the paper is organized as follows. Section 2 defines data-race-free-1. Sections 3 and 4 compare data-race-free-1 with the weak ordering, release consistency, VAX, and data-race-free-0 models from the viewpoint of a programmer and hardware designer respectively. Section 5 relates data-race-free-1 to other models. Section 6 concludes the paper.

2. The Data-Race-Free-1 Memory Model

Section 2.1 first clarifies common terminology that will be used throughout the paper and then informally motivates the data-race-free-1 memory model. Section 2.2 gives the formal definition of data-race-free-1. Data-race-free-1 is an extension of our earlier model data-race-free-0 [1].

2.1. Terminology and Motivation for Data-Race-Free-1

The rest of the paper assumes the following terminology. The terms *system*, *program*, and *operations* (as in definition 1.1 of sequential consistency) can be used at several levels. This paper discusses memory models at the lowest level, where the system is the machine hardware, a program is a set of machine-level instructions, and an operation is a memory operation that either reads a memory location (a *read* operation) or modifies a memory location (a *write* operation) as part of the machine instructions of the program. The *program order* for an execution is a partial order on the memory operations of the execution imposed by the program text [27]. The *result* of an execution refers to the values returned by the read operations in the execution. A *sequentially consistent execution* is an execution that could occur on sequentially consistent hardware. Two memory operations *conflict* if at least one of them is a write and they access the same location [27].

The motivation for data-race-free-1, which is similar to that for weak ordering, release consistency, the VAX model and data-race-free-0, is based on the following observations made in [1].[3] Assuming processors maintain uniprocessor data and control dependencies, sequential consistency can be violated only when two or more processors interact through memory operations on common locations. These interactions can be classified as *data* memory operations and *synchronization* memory operations. Data operations are usually more frequent and involve reading and writing of data. Synchronization operations are usually less frequent and are used to order conflicting data operations from different processors. For example, in the implementation of a critical section using semaphores, the test of the semaphore and the unset or clear of the semaphore are synchronization operations, while the reads and writes in the critical section are data operations.

Additionally, synchronization operations can be characterized as paired acquire and release synchronization operations or as unpaired synchronization operations as follows. (The characterization is similar to that used for properly labeled programs for release consistency [11]; Section 3 discusses the differences.) In an execution, consider a write and a read synchronization operation to the same location, where the read returns the value of the write, and the value is used by the reading processor to conclude the completion of all memory operations of the writing processor that were before the write in the program. In such an interaction, the write synchronization operation is called a *release*, the read synchronization operation is called an *acquire*, and the release and acquire are said to be *paired* with each other. A synchronization operation is *unpaired* if it is not paired with any other synchronization operation in the execution. For example, consider an implementation of a critical section using semaphores, where the semaphore is tested with a test&set instruction and is cleared with an unset instruction. The write due to an unset is paired with the test that returns the unset value; the unset write is a release operation

3. The observations are paraphrased from [1] with the permission of the IEEE.

and the test read is an acquire operation because the unset value returned by the test is used to conclude the completion of the memory operations of the previous invocation of the critical section. The write due to a set of a test&set and a read due to the test of a test&set that returns the set value are unpaired operations; such a read is not an acquire and the write is not a release because the set value does not communicate the completion of any previous memory operations.

As will be illustrated by Section 4, it is possible to ensure sequential consistency by placing most hardware restrictions only on the synchronization operations. Further, of the synchronization operations, the paired operations require more restrictions. Thus, if hardware could distinguish the type of an operation, it could complete data operations faster than all the other operations, and unpaired synchronization operations faster than the paired synchronization operations, without violating sequential consistency. A data-race-free-1 system gives programmers the option of distinguishing the above types of operations to enable higher performance.

2.2. Definition of Data-Race-Free-1

Section 2.1 informally characterized memory operations based on the function they perform, and indicated that by distinguishing memory operations based on this characterization, higher performance can be obtained without violating sequential consistency. This section first discusses how the memory operations can be distinguished based on their characterization on a data-race-free-1 system, and then gives the formal criterion for when the operations are distinguished correctly for data-race-free-1. The section concludes with the definition of the data-race-free-1 memory model.

Data-race-free-1 does not impose any restrictions on how different memory operations may be distinguished. One option for distinguishing data operations from synchronization operations is for hardware to provide different instructions that may be used for each type of operation. For example, only special instructions such as Test&Set and Unset may be used to generate synchronization operations. Alternatively, only operations to certain memory-mapped locations may be distinguished as synchronization operations. One way of distinguishing between paired and unpaired synchronization operations is for hardware to provide special instructions for synchronization operations and a static *pairable* relation on those instructions; a write and a read in an execution are distinguished by the hardware as paired release and acquire if they are generated by instructions related by the pairable relation, and if the read returns the value of the write. Figure 2 gives examples of different instructions and the pairable relation, and illustrates their use.

The following discusses when a programmer distinguishes operations correctly for data-race-free-1. If the operations are distinguished exactly according to their function outlined in Section 2.1, then the distinction is indeed correct. However, data-race-free-1 does not require a programmer to distinguish operations to match their

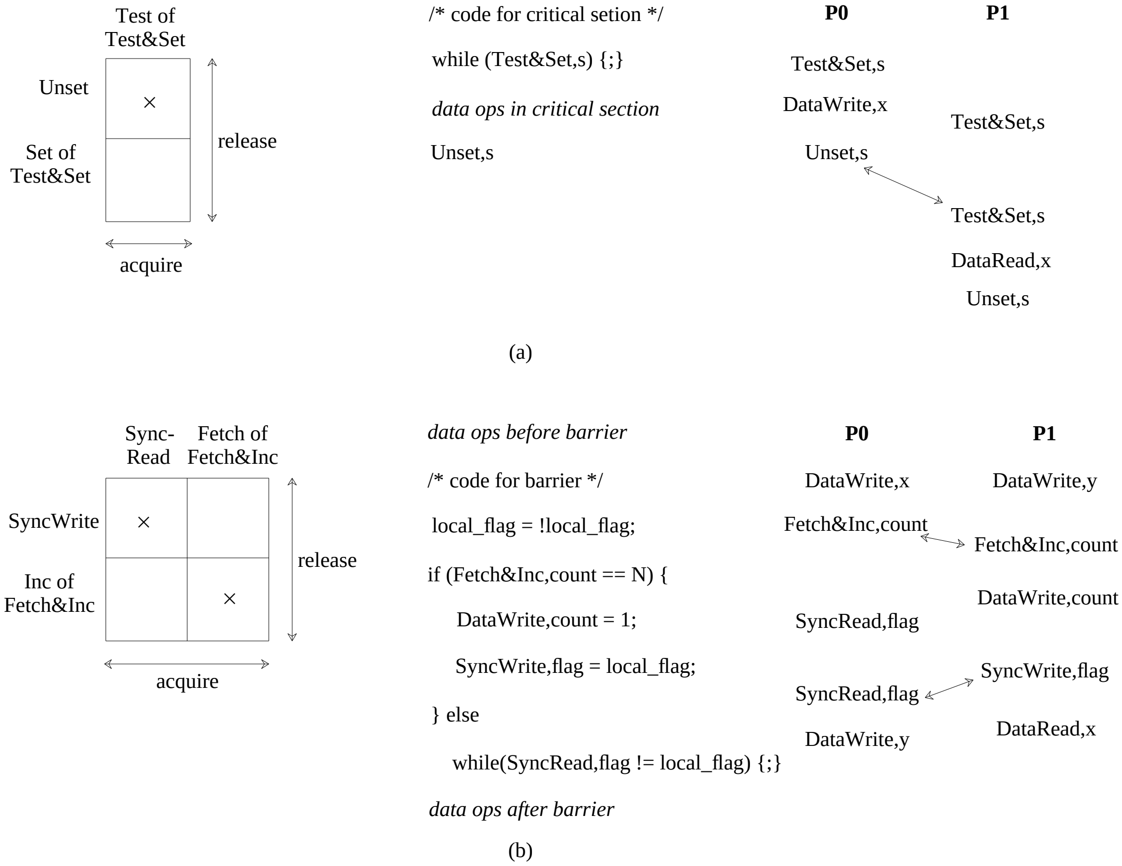


Figure 2. Synchronization instructions and the pairable relation for different systems.

Figures 2a and 2b represent two systems with different sets of instructions that can be used for synchronization operations. For each system, the figure shows the different synchronization operations and the pairable relation, along with programs and executions that use these operations. The table in each figure lists the read synchronization operations (potential acquires) horizontally, and the write synchronization operations (potential releases) vertically. A '×' indicates that the synchronization operations of the corresponding row and column are pairable; they will be paired in an execution if the read returns the value written by the write in that execution. The executions occur on sequentially consistent hardware and their operations execute in the order shown. op,x denotes an operation op on location x. DataRead and DataWrite denote data operations. The Test&Set and Fetch&Inc [17] instructions are defined to be atomic instructions. Their read and write operations are represented together as Test&Set,x or Fetch&Inc,x. Paired operations are connected with arrows.

Figure 2a shows a system with the Test&Set and Unset instructions, which are useful to implement a critical section. A Test&Set atomically reads a memory location and updates it to the value 1. An Unset updates a memory location to the value 0. A write due to an Unset and a read due to a Test&Set are pairable. The figure shows code for a critical section and its execution involving two processors.

Figure 2b shows a system with the Fetch&Inc [17], SyncWrite and SyncRead instructions, which are useful to implement a barrier. Fetch&Inc atomically reads and increments a memory location, SyncWrite is a synchronization write that updates a memory location to the specified value, and SyncRead is a synchronization read of a memory location. A write due to a Fetch&Inc is pairable with a read due to another Fetch&Inc and a write due to a SyncWrite is pairable with a read due to a SyncRead. Also shown is code where N processors synchronize on a barrier [23], and its execution for N = 2. The variable local_flag is implemented in a local register of the processor and operations on it are not shown in the execution.

function exactly. In the absence of precise knowledge regarding the function of an operation, a programmer can conservatively distinguish an operation as a synchronization operation even if the operation actually performs the function of a data operation. Sequential consistency will still be guaranteed although the full performance potential of the system may not be exploited. Henceforth, the characterization of an operation will be the one distinguished by the programmer (which may be different from that based on the actual function the operation performs). For example, an operation that is actually a data operation, but for which the programmer uses a synchronization instruction, will be referred to as a synchronization operation.

Intuitively, operations are distinguished correctly for data-race-free-1 if sufficient synchronization operations are distinguished as releases and acquires. The criteria for sufficiency is that if an operation is distinguished as data, then it should not be involved in a race; i.e, the program should be data-race-free. The notion of a data race is formalized by defining a *happens-before-1* relation for every execution of a program as follows.

The happens-before-1 relation for an execution is a partial order on the memory operations of the execution. Informally, happens-before-1 orders two operations initiated by different processors only if paired release and acquire operations execute between them. Definition 2.2 formalizes this intuition by using the program order and the synchronization-order-1 relations (Definition 2.1).

Definition 2.1: In an execution, memory operation S_1 is ordered before memory operation S_2 by the synchronization-order-1 relation if and only if S_1 is a release operation, S_2 is an acquire operation and S_1 and S_2 are paired with each other.

Definition 2.2: The happens-before-1 relation for an execution is the irreflexive transitive closure of the program order and synchronization-order-1 relations for the execution.

The definitions of a data race, a data-race-free program and the data-race-free-1 model follow.

Definition 2.3: A *data race* in an execution is a pair of conflicting operations, at least one of which is data, that is not ordered by the happens-before-1 relation defined for the execution. An execution is *data-race-free* if and only if it does not have any data races. A program is *data-race-free* if and only if all its sequentially consistent executions are data-race-free.

Definition 2.4: Hardware obeys the $data-race-free-1$ memory model if and only if the result of every execution of a data-race-free program on the hardware can be obtained by an execution of the program on sequentially consistent hardware.

Figures 3a and 3b illustrate executions that respectively exhibit and do not exhibit data races. The execution in Figure 3a is an implementation of the critical section code in Figure 2a, except that the programmer used a data write operation instead of the Unset synchronization operation for P_0's write on *s*. Therefore, happens-before-1 does not order P_0's write on *x* and P_1's read on *x*. Since the write and read on *x* conflict and are both data operations, they form a data race. For similar reasons, P_0's data write on *s* forms a data race with P_1's test, set and data write on *s*. Figure 3b shows an execution of the barrier code of Figure 2b. The execution is data-race-free because happens-before-1 orders all conflicting pairs of operations, where at least one of the pair is data.

Note that the execution of Figure 3b does not use critical sections and therefore data-race-free-1 does not require that all sharing be done through critical sections. Also note that in programs based on asynchronous algorithms [7], some operations access data, but are not ordered by synchronization. For such programs to be data-race-free, these operations also need to be distinguished as synchronization operations.

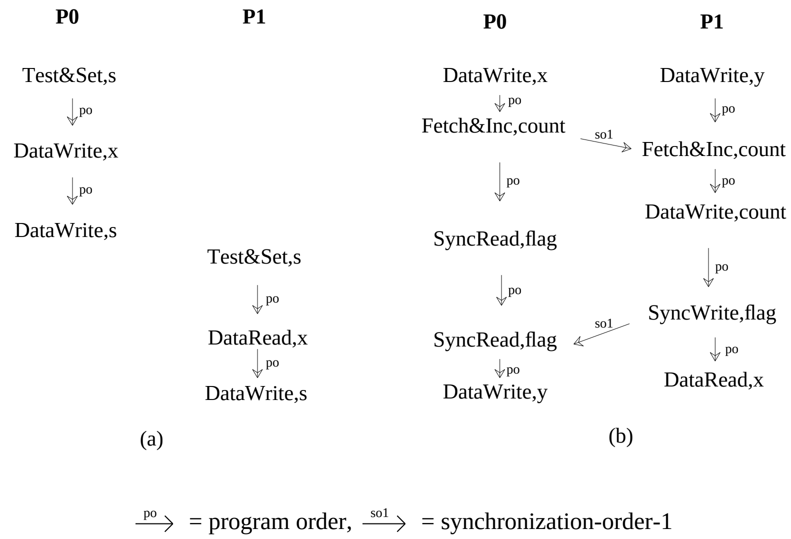


Figure 3. Executions that (a) exhibit and (b) do not exhibit data races.

As discussed in Section 2.1, the definition of data-race-free-1 assumes a program that uses machine instructions and hardware-defined synchronization primitives. However, programmers using high-level parallel programming languages can use data-race-free-1 by extending the definition of data-race-free to high-level programs (as discussed for data-race-free-0 in [1]). The extension is straightforward, but requires high-level parallel languages to provide special constructs for synchronization, e.g., semaphores, monitors, fork-joins, and task rendezvous. Data-race-free-1 does not place any restrictions on the high-level synchronization mechanisms. It is the responsibility of the compiler to ensure that a program that is data-race-free at the high-level compiles into one that is data-race-free at the machine-level, ensuring sequential consistency to the programmer.

3. Data-Race-Free-1 vs. Weak Ordering, Release Consistency, the VAX Model, and Data-Race-Free-0 for Programmers

This section compares the data-race-free-1 memory model to weak ordering, release consistency, the VAX, and data-race-free-0 from a programmer's viewpoint. As stated earlier, the central assumption of this work is that most programmers prefer to reason with sequential consistency. For such programmers, data-race-free-1 provides a simple model: if the program is data-race-free, then hardware will appear sequentially consistent.

Both weak ordering and the VAX memory model state that programs have to obey certain conditions for hardware to be well-behaved. However, sometimes further interpretation may be needed to deduce whether a program obeys the required conditions (as in the concurrent readers case of Section 1), and how the hardware will behave for programs that obey the required conditions. Data-race-free-1 expresses both these aspects more explicitly and formally than weak ordering and the VAX: data-race-free-1 states that a program should be data-race-free, and hardware appears sequentially consistent to programs that are data-race-free.

Data-race-free-0 and release consistency provide a formal interface for programmers. Data-race-free-1 provides a similar interface with a few minor differences. The programs for which data-race-free-0 ensures sequential consistency are also called data-race-free programs [1]. The difference is that data-race-free-0 does not distinguish between different synchronization operations; it effectively pairs all conflicting synchronization operations depending on the order in which they execute. This distinction does not significantly affect programmers, but can be exploited by hardware designers.

The programs for which release consistency ensures sequential consistency are called properly labeled programs [11]. All data-race-free programs are properly labeled, but there are some properly labeled programs that are not data-race-free (as defined by Definition 2.4) [15]. The difference is minor and arises because properly labeled programs have a less explicit notion of pairing. They allow conflicting data operations to be ordered by operations (nsyncs) that correspond to the nonpairable synchronization operations of data-race-free-1. Although a memory model that allows all hardware that guarantees sequential consistency to properly labeled programs has not been formally described, such a model would be similar to data-race-free-1 because of the similarity between data-race-free and properly labeled programs.

A potential disadvantage of data-race-free-1 relative to weak ordering and release consistency is for programmers of asynchronous algorithms that do not rely on sequential consistency for correctness [7]. Weak ordering and release consistency provide such programmers the option of reasoning with their explicit hardware conditions and writing programs that are not data-race-free, but work correctly and possibly faster. Data-race-free-1 is based on the assumption that programmers prefer to reason with sequential consistency. Therefore, it does not restrict the behavior of hardware for a program that is not data-race-free. Nevertheless, for maximum performance, programmers of asynchronous algorithms could deal directly with specific implementations of data-race-free-1. This would entail some risk of portability across other data-race-free-1 implementations, but would enable future faster implementations for the other, more common programs.

To summarize, for programmers, data-race-free-1 is similar to release consistency and data-race-free-0, but provides a more explicit and formal interface than weak ordering and the VAX model. Previous work discusses how the requirement of data-race-free programs for all the above models is not very restrictive for programmers [1, 11], and how data races [2] or violations of sequential consistency due to data races [14] may be dynamically detected with these models.

4. Data-Race-Free-1 vs. Weak Ordering, Release Consistency, the VAX Model, and Data-Race-Free-0 for Hardware Designers

This section compares data-race-free-1 to weak ordering, release consistency, the VAX model, and data-race-free-0 from a hardware designer's viewpoint. It first shows that data-race-free-1 unifies the four models for a hardware designer because any implementation of weak ordering, release consistency, the VAX model, or data-race-free-0 obeys data-race-free-1 (Section 4.1). It then shows that data-race-free-1 is less restrictive than weak ordering, release consistency, and data-race-free-0 for a hardware designer because data-race-free-1 allows an implementation not allowed by weak ordering, release consistency, or data-race-free-0 (Section 4.2).

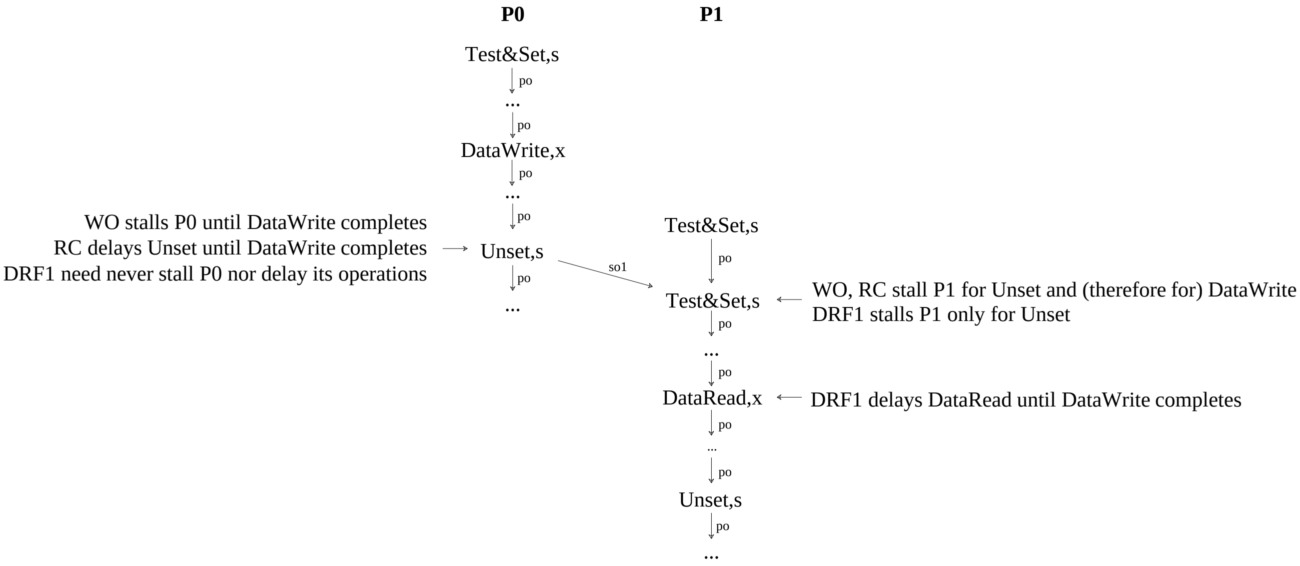
4.1. Data-Race-Free-1 Unifies Weak Ordering, Release Consistency, the VAX Model, and Data-Race-Free-0 for Hardware Designers

For a hardware designer, data-race-free-1 unifies release consistency, data-race-free-0, weak ordering, and the VAX model because any implementation of any of the four models obeys data-race-free-1. Specifically,

- all implementations of release consistency obey data-race-free-1 because, as discussed in Section 3, all implementations of release consistency ensure sequential consistency to all data-race-free programs;
- all implementations of data-race-free-0 obey data-race-free-1 because, again as discussed in Section 3, all implementations of data-race-free-0 ensure sequential consistency to all data-race-free programs;
- all implementations of weak ordering obey data-race-free-1 because our earlier work shows that all implementations of weak ordering obey data-race-free-0 [1], and from the above argument, all implementations of data-race-free-0 obey data-race-free-1;
- data-race-free-1 formalizes the VAX model; therefore, all implementations of the VAX model obey data-race-free-1.

4.2. Data-Race-Free-1 is Less Restrictive than Weak Ordering, Release Consistency, or Data-Race-Free-0 for Hardware Designers

Data-race-free-1 is less restrictive for a hardware designer to implement than either weak ordering, release consistency, or data-race-free-0 because data-race-free-1 allows an implementation that is not allowed by weak ordering, release consistency, or data-race-free-0. Figure 4 motivates such an implementation. The figure shows part of an execution in which two processors execute the critical section code of Figure 2a. Processors P_0 and P_1 Test&Set *s* until they succeed, execute data operations (including one on location *x*), and finally Unset *s*. The critical section code is data-race-free; therefore, its executions on a data-race-free-1 implementation should appear sequentially consistent. In the execution of Figure 4, P_0's Test&Set succeeds first. Therefore, P_1's Test&Set succeeds only when it returns the value written by P_0's Unset. Thus, to appear sequentially consistent, P_1's data read of *x* should return the value written by P_0's data write of *x*. Figure 4 shows how implementations of weak ordering, release consistency, and data-race-free-1 can achieve this.



$\xrightarrow{po}$ = program order, $\xrightarrow{so1}$ = synchronization-order-1

WO = weak ordering, RC = release consistency, DRF1 = data-race-free-1

Figure 4. Implementations of memory models.

Both weak ordering and release consistency require P_0 to delay the execution of its Unset until P_0's data write completes (i.e., is seen by all processors). However, this delay is not necessary to maintain sequential consistency (as also observed by Zucker [28]), and it is not imposed by the implementation proposal for data-race-free-1 described next. Instead, the implementation maintains sequential consistency by requiring that P_0's data write on *x* completes before P_1 executes its data read on *x*. It achieves this by ensuring that (i) when P_1 executes its Test&Set, P_0 notifies P_1 about its incomplete write on *x*, and (ii) P_1 delays its read on *x* until P_0's write on *x* completes.

With the new optimization, P_0 can execute its Unset earlier and P_1's Test&Set can succeed earlier than with weak ordering or release consistency. Thus, P_1's reads and writes following its Test&Set (by program order) that do not conflict with previous operations of P_0 will also complete earlier. Operations such as the data read on *x* that conflict with previous operations of P_0 may be delayed until P_0's corresponding operation completes. Nevertheless, such operations can also complete earlier than with weak ordering and release consistency. For example, if P_1's read on *x* occurs late enough in the program, P_0's write may already be complete before P_1 examines the read; therefore, the read can proceed without any delay. Recently, an implementation of release consistency has been proposed that uses a rollback mechanism to let a processor conditionally execute its reads following its acquire (such as P_1's Test&Set) before the acquire completes [12]; our optimization will benefit such implementations also because it allows the writes following the acquire to be issued and completed earlier, and lets the reads following the acquire to be committed earlier.

The data-race-free-1 implementation differs from data-race-free-0 implementations because data-race-free-1 distinguishes between the Unset and Test&Set synchronization operations and can take different actions for each; data-race-free-0 does not make such distinctions.

Section 4.2.1 describes a sufficient condition for implementing data-race-free-1 based on the above motivation. Section 4.2.2 gives a detailed implementation proposal based on these conditions.

4.2.1. Sufficient Conditions for Data-Race-Free-1

Hardware obeys the data-race-free-1 memory model if the result of any execution of a data-race-free program on the hardware can be obtained by a sequentially consistent execution of the program. The result of an execution is the set of values its read operations return (Section 2.1). The value returned by a read is the value from the write (to the same location) that was seen last by the reading processor. Thus, the value returned by a read depends on the order in which the reading processor sees its read with respect to writes to the same location; i.e., the order in which a processor sees conflicting operations. Thus, hardware is data-race-free-1 if it obeys the following conditions.

> **Data-Race-Free-1 Conditions:** Hardware is data-race-free-1 if for every execution, *E*, of a data-race-free program on the hardware, (i) the operations of execution *E* are the same as those of some sequentially consistent execution of the program, and (ii) the order in which two conflicting operations are seen by a processor in execution *E* is the same as in that sequentially consistent execution.

(A processor *sees* a write when a read executed by the processor to the same location as the write will return the value of that or a subsequent write. A processor *sees* a read when the read returns its value. These notions are similar to those of ‘‘performed with respect to a processor’’ and ‘‘performed’’ [9].)

The following gives three requirements (data, synchronization, and control) that are together sufficient for hardware to satisfy the data-race-free-1 conditions, and therefore to obey data-race-free-1.

The *data requirement* pertains to all pairs of conflicting operations of a data-race-free program, where at least one of the operations is a data operation. In an execution on sequentially consistent hardware, such a pair of operations is ordered by the happens-before-1 relation of the execution, and is seen by all processors in that happens-before-1 order. The data requirement for an execution on data-race-free-1 hardware is that all such pairs of operations continue to be seen by all processors in the happens-before-1 order of the execution. This requirement ensures that in Figure 4, P_1 sees P_0's write of x before the read of x. Based on the discussion of Figure 4, the data requirement conditions below meet the data requirement for a pair of conflicting operations from different processors. For conflicting operations from the same processor, it is sufficient to maintain intra-processor data dependencies. The conditions below assume these are maintained.

In the rest of this section, *preceding* and *following* refer to the ordering by program order. An operation, either synchronization or data, *completes* (or performs [9]) when it is seen (as defined above) by all processors.

> **Data Requirement Conditions:** Let *Rel* and *Acq* be release and acquire operations issued by processors P_{rel} and P_{acq} respectively. Let *Rel* and *Acq* be paired with each other.
>
> *Pre-Release Condition* - When P_{rel} issues *Rel*, it *remembers* the operations preceding *Rel* that are incomplete.
>
> *Release-Acquire Condition* - (i) Before *Acq* completes, P_{rel} transfers to P_{acq} the addresses and identity of all its *remembered* operations. (ii) Before *Acq* completes, *Rel* completes and all operations transferred to P_{rel} (on P_{rel}'s acquires preceding *Rel*) complete.
>
> *Post-Acquire Condition* - Let *Acq* precede *Y* (by program order) and let the operation *X* be transferred to P_{acq} on *Acq*. (i) Before *Y* is issued, *Acq* completes. (ii) If *X* and *Y* conflict, then before *Y* is issued, *X* completes.

The data requirement conditions can be proved correct by showing that they ensure that if *X* and *Y* are conflicting operations from different processors and happens-before-1 orders *X* before *Y*, then *X* completes before any processor sees *Y*. This implies that all processors see *X* before *Y*, meeting the data requirement. For the execution in Figure 4, the pre-release condition ensures that when P_0 executes its Unset, it remembers that DataWrite,x is incomplete. The release-acquire condition ensures that when P_1 executes its successful Test&Set, P_0 transfers the address of x to P_1. The post-acquire condition ensures that P_1 detects that it has to delay DataRead,x until DataWrite,x completes and enforces the delay. Thus, DataRead,x returns the value written by DataWrite,x.

Besides the data requirement, the data-race-free-1 conditions also require that the order in which two conflicting synchronization operations are seen by a processor is as on sequentially consistent hardware. This is the *synchronization requirement*. The data and synchronization requirements would suffice to satisfy the data-race-free-1 conditions if they also guaranteed that for any execution, *E*, on hardware that obeyed these

-- --

requirements, there is some sequentially consistent execution with the same operations, the same happens-before-1, and the same order of execution of conflicting synchronization operations as E. In the absence of control flow operations (such as branches), the above is automatically ensured. In the presence of control flow operations, however, an extra requirement, called the *control requirement*, is needed to ensure the above [3].

Weak ordering, release consistency, and all proposed implementations of data-race-free-0 satisfy the synchronization requirement explicitly and the control requirement implicitly (by requiring ‘‘uniprocessor control dependencies’’ to be maintained). Since the key difference between implementations of the earlier models and the new implementation of data-race-free-1 is in the data requirement, the following describes an implementation proposal only for the data requirement conditions. In [3], we formalize the above three requirements and give explicit conditions for the synchronization and control requirements. A conservative way to satisfy the synchronization requirement is for a processor to also stall the issue of a synchronization operation until the completion of preceding synchronization operations and the write operations whose values are returned by preceding synchronization read operations. A conservative way to satisfy the control requirement is for a processor to also block on a read that controls program flow until the read completes.

Note that further optimizations on the data requirement conditions and on the implementation of the following section are possible [3]. For example, for the release-acquire condition, the acquire can complete even while operations transferred to the releasing processor are incomplete, as long as the releasing processor transfers the identity of those incomplete operations to the acquiring processor. For the post-acquire condition, it is not necessary to delay an operation (Y) following an acquire until a conflicting operation (X) transferred to the acquiring processor completes. Instead, it is sufficient to delay Y only until X is seen by the acquiring processor, as long as a mechanism (such as a cache-coherence protocol) ensures that all writes to the *same* location are seen in the same order by all processors. Thus, the releasing processor can also transfer the values to be written by its incomplete writes. Then reads following an acquire can use the transferred values and need not be delayed.

4.2.2. An Implementation Proposal for Data-Race-Free-1 that does not obey Weak Ordering, Release Consistency, or Data-Race-Free-0

This section describes an implementation proposal for the data requirement conditions. The proposal assumes an arbitrarily large shared-memory system in which every processor has an independent cache and processors are connected to memory through an arbitrary interconnection network. The proposal also assumes a directory-based, writeback, invalidation, ownership, hardware cache-coherence protocol, similar in most respects to those discussed by Agarwal et al. [4]. One significant feature of the protocol is that invalidations sent on a write to a line in read-only or shared state are acknowledged by the invalidated processors.

The cache-coherence protocol ensures that (a) all operations are eventually seen by all processors, (b) writes to the same location are seen in the same order by all processors, and (c) a processor can detect when an operation it issues is *complete*. For (c), most operations complete when the issuing processor receives the requested line in its cache. However, a write (data or synchronization) to a line in read-only or shared state completes when all invalidated processors send their acknowledgements. (Either the writing processor may directly receive the acknowledgements, or the directory may collect them and then forward a single message to the writing processor to indicate the completion of the write.)

The implementation proposal involves adding the following four features to a uniprocessor-based processor logic and the base cache-coherence logic mentioned above. (Tables 1 and 2 summarize these features.)

- Addition of three buffers per processor -- incomplete, reserve, and special (Table 1),
- Modification of issue logic to delay the issue of or stall on certain operations (Table 2(a)),
- Modification of cache-coherence logic to allow a processor to retain ownership of a line in the processor's reserve buffer and to specially handle paired acquires to such a line (Table 2(b)),
- a new processor-to-processor message called ‘‘empty special buffer’’ (Table 2(c)).

The discussion below explains how the above features can be used to implement the pre-release, release-acquire, and post-acquire parts of the data requirement conditions. (Recall that ‘‘preceding’’ and ‘‘following’’ refer to the ordering by program order.)

For the pre-release condition, a processor must remember which operations preceding its releases are incomplete. For this, a processor uses its *incomplete buffer* to store the address of all its incomplete data operations. A release is not issued until all preceding synchronization operations complete (to prevent deadlock) and all preceding data operations are issued. Thus, the incomplete buffer remembers all the operations required by the pre-release condition. (To distinguish between operations preceding and following a release, entries in the incomplete buffer may be tagged or multiple incomplete buffers may be used.)

For the release-acquire condition, an acquire cannot complete until the following have occurred regarding the release paired with the acquire: (a) release is complete, (b) all operations received by the releasing processor on its acquires preceding the release are complete, and (c) the releasing processor transfers to the new acquiring processor the addresses of all incomplete operations preceding the release. For this purpose, every processor uses a *reserve buffer* to store the processor's releases for which the above conditions do not hold. On a release (which is a write operation), the releasing processor procures ownership of the released line. The processor does not give up its ownership while the address of the line is in its reserve buffer. Consequently, the cache-coherence protocol forwards subsequent requests to the line, including acquires that will be paired with the release, to the releasing

Buffer	Contents	Purpose
Incomplete	Incomplete data operations (of this processor)	Used to remember incomplete operations (of this processor) preceding a release (of this processor).
Reserve	Releases (of this processor) for which there are incomplete operations	Used to remember releases (of this processor) that may cause future paired acquires (of other processors) to need special attention.
Special	Incomplete operations (of another processor) received on an acquire (by this processor)	Used to identify if an operation (of this processor) requires special action due to early completion of acquire (of this processor).

(a) Contents and purpose of buffers

Buffer	Insertions		Deletions	
	Event	*Entry Inserted*	*Event*	*Entry Deleted*
Incomplete	Data miss	Address of data operation	Data miss completes	Address of data operation
Reserve	Release issued	Address of release operation	Release completes, operations preceding release complete (i.e., deleted from incomplete buffer), and special buffer empties	Address of release operation
Special	Acquire completes	Addresses received on acquire	‘‘Empty special buffer’’ message arrives	All entries

(b) Insertion and deletion actions for buffers

Table 1. Key buffers for aggressive implementation of data-race-free-1.

processor. The releasing processor can now stall the acquires paired with the release until conditions (a), (b), and (c) above are met.

Table 2(b) gives the details of how the base cache-coherence logic can be modified to allow a releasing processor to retain ownership of the released line in its reserve buffer, and to service acquires paired with the release only when (a), (b), and (c) above are met. To retain ownership of a released line, the releasing processor stalls release operations from other processors to the same line and performs a *remote service* for other external requests to the same line. The remote service mechanism allows the releasing processor to service the requests of other

Operation	Address in Special Buffer?	Action
Data or unpaired synchronization	No	Process as usual.
Release	No	Issue after all previous operations are issued and all previous synchronization operations complete.
Acquire	No	Issue after special buffer empties and stall until acquire completes.
Any	Yes	Stall or delay issue of only this operation until special buffer empties.

(a) Modification to issue logic

Request	Address in Reserve Buffer?	Action
Requests by this processor		
Any	No	Process as usual.
Any read or write	Yes	Process as usual.
Cache line replacement	Yes	Stall processor until address is deleted from reserve buffer.
Requests from other processors forwarded to this processor		
Any	No	Process as usual.
Release	Yes	Stall request until address is deleted from reserve buffer.
Acquire	Yes	Stall request until special buffer empties and paired release (in reserve buffer) completes, send to acquiring processor the released line and entries of incomplete buffer tagged as preceding the release, request acquiring processor to not cache the line, inform directory that this processor is retaining ownership.
Data or unpaired synchronization	Yes	If read request, send line to other processor; if write request, update line in this processor's cache and send acknowledgement to other processor; request other processor to not cache the line; inform directory that this processor is retaining ownership.

(b) Modification to cache-coherence logic at processor

Event	Message
All incomplete buffer entries corresponding to a release deleted	Send ‘‘empty special buffer’’ message to processors that executed acquires paired with release.

(c) New processor-to-processor message

Table 2. Aggressive implementation of data-race-free-1.

processors without allowing those processors to cache the line. The mechanisms of stalling operations for an external release and remote service for other external operations are both necessary. This is because stalling data operations can lead to deadlock and servicing external release operations remotely would not let the new releasing processors procure ownership of the line as required for the release-acquire condition. Meeting conditions (a), (b), and (c) above requires the processor to wait for its release to complete and its special buffer to empty, and to transfer contents of its incomplete buffer to the acquiring processor.

For the post-acquire condition, a processor must (a) stall on an acquire until it completes, and (b) delay a following operation until the completion of any conflicting operation transferred to it on the acquire. For this purpose, a processor uses a *special buffer* to save all the information transferred to it on an acquire. If a following operation conflicts with an operation stored in the special buffer, the processor can either (a) stall or (b) delay only this operation, until it receives an ‘‘empty special buffer’’ message from the releasing processor. The releasing processor sends the ‘‘empty special buffer’’ message when it deletes the address of the release paired with the acquire from its reserve buffer. For simplicity, an acquiring processor can also stall on the acquire until its special buffer empties to avoid the complexity of having to delay an operation for incomplete operations of multiple processors.

This completes the implementation proposal for the data requirement conditions, assuming a process runs uninterrupted on the same processor. To handle context switches correctly, a processor must stall before switching until the various buffers mentioned above empty. Overflow of the above buffers can also be handled by making a processor stall until an entry is deleted from the relevant buffer.

The above proposal never leads to deadlock or livelock as long as the underlying cache-coherence protocol is implemented correctly, and messages are not lost in the network (or a time-out that initiates a system clean-up is generated on a lost message). Specifically, the above proposal never stalls a memory operation indefinitely since (i) the proposal never delays the completion of issued data operations, and (ii) the proposal delays an operation only if certain issued data operations are incomplete. Thus, the above proposal does not lead to deadlock or livelock.

5. Data-Race-Free-1 vs. Other Models

Previous sections have shown how the data-race-free-1 memory model unifies weak ordering, release consistency, the VAX model, and data-race-free-0. This section first summarizes other memory models proposed in the literature, and then examines how data-race-free-1 relates to them.

The IBM 370 memory model [19] guarantees that except for a write followed by a read to a different location, operations of a single processor will appear to execute in program order, and writes will appear to execute atomically. The 370 also provides serialization operations. Before executing a serialization operation, a processor completes all operations that are before the serialization operation according to program order. Before executing any nonserialization operation, a processor completes all serialization operations that are before that nonserialization operation according to program order. The processor consistency [11, 16], PRAM [22] and total store ordering [25] models ensure that writes of a given processor appear to execute in the same order to all other processors. The models mainly differ in whether a write appears to become visible to all other processors simultaneously or at different times. The partial store ordering model [25] is similar to total store ordering except that it orders writes by a processor only if they are separated by a store barrier operation. The model known as release consistency with processor-consistent special operations [11] is similar to release consistency with sequentially consistent special operations except that it requires special operations (syncs and nsyncs) to be processor-consistent. The concurrent-consistency model [26] ensures sequential consistency to all programs except those ‘‘which explicitly test for sequential consistency or take access timings into consideration.’’ The slow memory model [18] requires that a read return the value of some previous conflicting write. After a value written by (say) processor P_i is read, the values of earlier conflicting writes by P_i cannot be returned. The causal memory model [5, 18] ensures that any write that causally precedes a read is observed by the read. Causal precedence is a transitive relation established by program order or due to a read that returns the value of a write.

Data-race-free-1 is based on the assumption that most programmers prefer to reason with sequential consistency. Concurrent consistency is the only model above that explicitly states when programmers can expect sequential consistency; however, the conditions that give sequential consistency seem ambiguous and are difficult to relate directly to data-race-free-1. The 370 model does not explicitly state when programmers can expect sequential consistency; however, the previous sections on data-race-free-1 can be used to determine a sufficient condition as follows. The serialization operations are analogous to the synchronization operations of weak ordering; therefore, the 370 appears sequentially consistent to data-race-free programs where serialization operations that access memory are interpreted as synchronization operations and every write serialization operation is pairable with every read serialization operation.

For the remaining models, it is difficult to determine exactly when programmers can expect sequential consistency. If the assumption that programmers prefer to reason with sequential consistency is true, then as stated, the above models are harder to reason with than data-race-free-1. In the future, we hope to specify the above models using the approach of data-race-free-1; i.e., specify the models in terms of a formal set of constraints on programs such that the hardware appears sequentially consistent to all programs that obey those constraints. We call this approach the *sequential consistency normal form*. We will investigate if such specifications provide greater insight and lead to more unifications.

6. Conclusions

Many programmers of shared-memory systems implicitly assume the model of sequential consistency for the shared memory. Unfortunately, sequential consistency restricts the use of many high performance uniprocessor optimizations. For higher performance, several alternate memory models have been proposed. Such models should (1) be simple to reason with and (2) provide high performance. We believe that most programmers prefer to reason with sequential consistency. Therefore, a way to satisfy the above properties is for a model to appear sequentially consistent to the most common programs and to give these programs the highest performance possible. The models of weak ordering, release consistency (with sequentially consistent special operations), the VAX, and data-race-free-0 are based on the common intuition that if programmers distinguish their data and synchronization operations, then correct execution can be guaranteed along with high performance. However, each model formalizes the intuition differently, and has different advantages and disadvantages with respect to the other models.

This paper proposed a memory model, data-race-free-1, that unifies weak ordering, release consistency, the VAX model and data-race-free-0, and retains the advantages of each of them. Hardware is data-race-free-1 if it appears sequentially consistent to all programs that are data-race-free. Data-race-free-1 unifies the four models by providing a programmer's view that is similar to that of the four models, and by permitting all hardware allowed by the four models. Compared to weak ordering, data-race-free-1 provides a more formal interface for programmers since it explicitly states when a program is correctly synchronized (data-race-free) and how hardware behaves for correctly synchronized programs (sequentially consistent). Also, data-race-free-1 is less restrictive than weak ordering for hardware designers since it allows an implementation that weak ordering does not allow. Compared to release consistency, data-race-free-1 is less restrictive for hardware designers since it allows an implementation that release consistency does not allow. Compared to the VAX model, data-race-free-1 provides a more formal interface since it explicitly states when a program is correctly synchronized and how hardware behaves for correctly synchronized programs. Compared to data-race-free-0, data-race-free-1 is less restrictive for hardware designers since it allows implementations to take different actions on different types of synchronization operations.

Acknowledgements

We are immensely grateful to Dr. Harold Stone, the editor, for his advice and patience through several revisions of this paper. We are also grateful to the anonymous referees for many comments and suggestions that have improved this work considerably. We thank Kourosh Gharachorloo for many insightful discussions on memory models and comments on earlier drafts of this paper. We also thank Vikram Adve, Brian Bershad, Allan Gottlieb, Ross Johnson, Alex Klaiber, Jim Larus, David Wood, and Richard Zucker for their valuable comments on earlier drafts of this paper.

References

1. S. V. ADVE and M. D. HILL, Weak Ordering - A New Definition, *Proc. 17th Annual Intl. Symp. on Computer Architecture*, May 1990, 2-14.
2. S. V. ADVE, M. D. HILL, B. P. MILLER and R. H. B. NETZER, Detecting Data Races on Weak Memory Systems, *Proc. 18th Annual Intl. Symp. on Computer Architecture*, May 1991, 234-243.
3. S. V. ADVE and M. D. HILL, Sufficient Conditions for Implementing the Data-Race-Free-1 Memory Model, Computer Sciences Technical Report #1107, University of Wisconsin, Madison, September 1992.
4. A. AGARWAL, R. SIMONI, M. HOROWITZ and J. HENNESSY, An Evaluation of Directory Schemes for Cache Coherence, *Proc. 15th Annual Intl. Symp. on Computer Architecture*, Honolulu, Hawaii, June 1988, 280-289.
5. M. AHAMAD, P. W. HUTTO and R. JOHN, Implementing and Programming Causal Distributed Shared Memory, College of Computing Technical Report GIT-CC-90-49, Georgia Institute of Technology, November 1990.
6. J. ARCHIBALD and J. BAER, Cache Coherence Protocols: Evaluation Using a Multiprocessor Simulation Model, *ACM Trans. on Computer Systems 4*, 4 (November 1986), 273-298.
7. R. DELEONE and O. L. MANGASARIAN, Asynchronous Parallel Successive Overrelaxation for the Symmetric Linear Complementarity Problem, *Mathematical Programming 42*, 1988, 347-361.
8. DEC., VAX Architecture Handbook, 1981.
9. M. DUBOIS, C. SCHEURICH and F. A. BRIGGS, Memory Access Buffering in Multiprocessors, *Proc. 13th Annual Intl. Symp. on Computer Architecture 14*, 2 (June 1986), 434-442.
10. M. DUBOIS and C. SCHEURICH, Memory Access Dependencies in Shared-Memory Multiprocessor, *IEEE Trans. on Software Engineering SE-16*, 6 (June 1990), 660-673.
11. K. GHARACHORLOO, D. LENOSKI, J. LAUDON, P. GIBBONS, A. GUPTA and J. HENNESSY, Memory Consistency and Event Ordering in Scalable Shared-Memory Multiprocessors, *Proc. 17th Annual Intl. Symp. on Computer Architecture*, May 1990, 15-26.
12. K. GHARACHORLOO, A. GUPTA and J. HENNESSY, Two Techniques to Enhance the Performance of Memory Consistency Models, *Proc. Intl. Conf. on Parallel Processing*, 1991, I355-I364.
13. K. GHARACHORLOO, A. GUPTA and J. HENNESSY, Performance Evaluation of Memory Consistency Models for Shared-Memory Multiprocessors, *Proc. 4th Intl. Conf. on Architectural Support for Programming Languages and Operating Systems*, 1991, 245-257.
14. K. GHARACHORLOO and P. B. GIBBONS, Detecting Violations of Sequential Consistency, *Proc. Symp. on Parallel Algorithms and Architectures*, July 1991, 316-326.
15. P. B. GIBBONS, M. MERRITT and K. GHARACHORLOO, Proving Sequential Consistency of High-Performance Shared Memories, *Proc. Symp. on Parallel Algorithms and Architectures*, July 1991, 292-303.
16. J. R. GOODMAN, Cache Consistency and Sequential Consistency, Computer Sciences Technical Report #1006, Univ. of Wisconsin, Madison, February 1991.

17. 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*, February 1983, 175-189.

18. P. W. HUTTO and M. AHAMAD, Slow Memory: Weakening Consistency to Enhance Concurrency in Distributed Shared Memories, *Proc. 10th Intl. Conf. on Distributed Computing Systems*, 1990, 302-311.

19. IBM, IBM System/370 Principles of Operation, Publication Number GA22-7000-9, File Number S370-01, May 1983.

20. D. KROFT, Lockup-Free Instruction Fetch/Prefetch Cache Organization, *Proc. Eighth Symp. on Computer Architecture*, May 1981, 81-87.

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

22. R. J. LIPTON and J. S. SANDBERG, PRAM: A Scalable Shared Memory, Technical Report CS-Tech. Rep.-180-88, Princeton University, September 1988.

23. J. M. MELLOR-CRUMMEY and M. L. SCOTT, Algorithms for Scalable Synchronization on Shared-Memory Multiprocessors, *ACM Transactions on Computer Systems*, February 1991, 21-65.

24. J. PATEL, Mulitprocessor Cache Memories, Seminar at Texas Instruments Research Labs (Dallas, Dec. 1983), Intel (Aloha, Oregon, April 1984), Digital Equipment (Hudson, Mass., June 1984), IBM (Yorktown, Oct. 1984), IBM (Poughkeepsie, Aug. 1986).

25. SUN, The SPARC Architecture Manual, Sun Microsystems Inc., No. 800-199-12, Version 8, January 1991.

26. 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.

27. 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.

28. R. N. ZUCKER, A Study of Weak Consistency Models, Dissertation Proposal, University of Washington, 1991.