*To Appear in the Proceedings of the 18th Annual International Symposium on Computer Architecture*

# Detecting Data Races on Weak Memory Systems†

*Sarita V. Adve, Mark D. Hill, Barton P. Miller, Robert H.B. Netzer*

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

ABSTRACT

For shared-memory systems, the most commonly assumed programmer's model of memory is sequential consistency. The weaker models of weak ordering, release consistency with sequentially consistent synchronization operations, data-race-free-0, and data-race-free-1 provide higher performance by guaranteeing sequential consistency to only a restricted class of programs - mainly programs that do not exhibit data races. To allow programmers to use the intuition and algorithms already developed for sequentially consistent systems, it is important to determine when a program written for a weak system exhibits no data races. In this paper, we investigate the extension of dynamic data race detection techniques developed for sequentially consistent systems to weak systems. A potential problem is that in the presence of a data race, weak systems fail to guarantee sequential consistency and therefore dynamic techniques may not give meaningful results. However, we reason that in practice a weak system will preserve sequential consistency at least until the ''first'' data races since it cannot predict if a data race will occur. We formalize this condition and show that it allows data races to be dynamically detected. Further, since this condition is already obeyed by all proposed implementations of weak systems, the full performance of weak systems can be exploited.

*Keywords:* data races, sequential consistency, weak ordering, release consistency, data-race-free-0, data-race-free-1.

† Research supported in part by National Science Foundation grants CCR-8815928, CCR-8902536 and MIPS-8957278, Office of Naval Research grant N00014-89-J-1222, and external research grants from A. T. & T. Bell Laboratories, Cray Research, Digital Equipment Corporation and Texas Instruments.

## 1. Introduction

The programmer's model of common shared-memory multiprocessor systems is a collection of sequential processors and a specification of the semantics of shared memory. Intuitively, a read of shared memory should return the value of the ''last'' write to the same address. A specification of shared memory semantics, called a *memory model* or *memory consistency model*, is necessary to determine exactly which writes could be ''last''.

The most commonly and often implicitly assumed memory model is *sequential consistency* [Lam79], a direct extension of the way memory is viewed in a multiprogrammed uniprocessor. An execution (of a program) on a shared-memory multiprocessor is sequentially consistent if reads return values such that all memory operations appear to have occurred in a sequential order, where the order of operations of an individual processor is as specified by its program. A system provides sequential consistency if it only allows sequentially consistent executions. Implementing sequential consistency in shared-memory multiprocessors, however, restricts the use of many performance enhancing features of uniprocessor systems, such as out-of-order instruction issue and completion, write buffers, and lockup-free caches [DSB86].

For higher performance, researchers have proposed alternative memory models [AdH90, AdH91, DSB86, GLL90, Goo91]. These models, however, do not provide sequential consistency to all programs. To allow programmers to use the intuition and algorithms already developed for sequential consistency, it is important to characterize the programs for which the models do provide sequential consistency.

The models of weak ordering (WO) [DSB86], release consistency with sequentially consistent synchronization operations ($RC_{sc}$) [GLL90], data-race-free-0 (DRF0) [AdH90], and data-race-free-1 (DRF1) [AdH91] are similar in the respect that they provide sequential consistency to all programs that exhibit no data races in any execution on sequentially consistent hardware (i.e., to *data-race-free* programs). Informally, two memory operations in an execution form a data race if at least one of them is a data operation (as opposed to a synchronization operation), at least one of them is a write (as opposed to a read), both access the same memory location, and they are not ordered by intervening synchronization operations. We would like to determine when programs for the above models are data-race-free, so programmers can assume sequential consistency when designing and debugging their programs. Henceforth, the systems specified by WO, $RC_{sc}$, DRF0 and DRF1 will collectively be referred to as the *weak systems* and an execution on a weak system will be referred to as a *weak execution*.

The problem of detecting data races is not unique to weak systems. Even on sequentially consistent systems, the presence of data races makes it hard to reason about a program and is usually considered to be a bug. Much current research in parallel programming has therefore been devoted to detecting data races in programs written for sequentially consistent systems. *Static techniques* perform a compile-time analysis of the program text to detect a superset of all possible data races that could potentially occur in all possible sequentially consistent executions of the program [BaK89, Tay83a]. In general, static analysis must be conservative and slow, because detecting data races is undecidable for arbitrary programs [Ber66] and is NP-complete for even very restricted classes of programs (e.g., those containing no branches) [Tay83b]. *Dynamic techniques*, on the other hand, use a tracing mechanism to detect whether a particular sequentially consistent execution of a program actually exhibited a data race [AlP87, ChM91, DiS90, HKM90, NeM90, NeM91]. While dynamic techniques provide precise information about a single execution, they provide little information about other executions, a serious disadvantage. For these reasons, the general consensus among researchers investigating data race detection is that tools should support both static and dynamic techniques in a complementary fashion [EmP88].

Rather than start from scratch, we seek to apply the data race detection techniques from sequentially consistent systems to weak systems. Static techniques can be applied to programs for weak systems unchanged, because they do not rely on executing the program. Dynamic techniques, however, depend on executing a program. If a program has no data races, then all executions of it on a weak system will be sequentially consistent, and the dynamic techniques will correctly conclude that no data races occurred. If a program has data races, on the other hand, an execution of it on a weak system may not be sequentially consistent. Applying the dynamic techniques to such an execution may produce unpredictable results.

In this paper we develop a hardware condition for weak systems that allows data races to be dynamically detected. The key observation we use is that practical implementations of weak systems give sequential consistency at least until the first data races (those not affected by others) because they cannot predict that a data race will occur. We formalize this observation with a condition that all executions on weak systems have a *sequentially consistent prefix* that extends to the first data races (or the end of the execution). With this condition, we show how to use a dynamic approach that either (a) correctly determines no data races occurred and concludes that the execution was sequentially consistent, or (b) identifies the first data races that could have occurred in a sequentially consistent execution with approximately the same accuracy as on a sequentially consistent system. Furthermore, since all implementations of WO and $RC_{sc}$ and all proposed implementations of DRF0 and DRF1 already exhibit the required sequentially consistent prefix, we argue that the new condition is obeyed for free in practice.

Alternatively, one could use dynamic techniques on weak systems by having weak systems support a slower sequentially consistent mode that is used for debugging. The results of this paper, however, show that such a slower mode is not necessary for detecting data races. Furthermore, we believe that our results also allow other debugging tools for sequentially consistent systems to be used unchanged on weak systems. If there are no data races in the execution, then the execution is sequentially consistent and other debugging tools can be directly applied. If there are data races, then the presence of the sequentially consistent prefix allows the tools to be applied to the part of the execution that contains the first bugs of the program.

The rest of the paper is organized as follows. Section 2 formalizes the notion of a data race and briefly reviews the weak models. Section 3 discusses the potential problems in applying the current dynamic techniques to weak systems, develops the hardware condition to overcome these problems, and explains why all proposed implementations of the weak systems already obey this condition. Section 4 shows how data races can be dynamically detected on a system obeying our condition. Section 5 discusses how the limitations of our dynamic technique on weak hardware are similar to those that occur on sequentially consistent hardware. Section 6 concludes the paper.

## 2. Data Races and Weak Models

In this section, we formalize the notion of a data race on sequentially consistent systems, and then briefly review the weak models.

### 2.1. Formalizing Data Races

We first clarify some terminology used throughout the paper. The term *program* refers to the program text (a set of machine instructions) and the input data. A machine instruction involves zero or more memory operations; each memory operation either reads a memory location (a read operation) or modifies a memory location (a write operation). Further, an operation is uniquely identified by the location it accesses and the part of the program in which it is specified; the value it reads or writes is not considered. For each of its executions, a program defines a partial order on the memory operations of that execution, called the *program order* (denoted by $\xrightarrow{po}$). Finally, two memory operations *conflict* if they access the same location and at least one of them is a write operation.

Memory operations can be partitioned into two groups: synchronization operations that are used to order events, and the more frequent data operations that read and write data. Often different instructions are used for the two groups. We will call an operation a *synchronization operation* if it is recognized by the hardware as meant for synchronization. For example, a synchronization operation could be a read or a write operation by a special instruction such as a Test&Set, or it could be an ordinary read or a write operation to a special location known to the hardware. Operations that are not recognized as synchronization by the hardware will be called *data operations*.

For a formal definition of a data race, we define a *happens-before-1* or $\xrightarrow{hb1}$ relation for every sequentially consistent execution of a program. Our definition is closely related to the ''happened-before'' relation defined by Lamport [Lam78] for message-passing systems. The $\xrightarrow{hb1}$ relation for a sequentially consistent execution is a partial order on its memory operations. Two operations initiated by the same processor are ordered by $\xrightarrow{hb1}$ according to program order. Two operations initiated by different processors are ordered by $\xrightarrow{hb1}$ only if there exist intervening synchronization operations. Further, with the synchronization operations that are allowed to be used for ordering, there should be associated certain semantics. We encapsulate these semantics by using the classification of release and acquire synchronization operations proposed in [GLL90] and a pairing relation as follows:

**Definition 2.1:** Two synchronization operations, $s_1$ and $s_2$, are *paired* iff the following are satisfied.

(1) Synchronization operation $s_1$ is a write and can be used by a processor to communicate the completion of all its previous memory operations (in program order) to a processor that subsequently executes operation $s_2$. Such a write is called a *release* operation.

(2) Synchronization operation $s_2$ is a read and can be used by a processor to conclude the completion of previous operations of a processor that executed an $s_1$ earlier in the execution. Such a read is called an *acquire* operation.

(3) Operations $s_1$ and $s_2$ access the same location and $s_2$ returns the value written by $s_1$.

For every system, the classification of synchronization operations into release and acquire operations that can potentially be paired can be determined by the semantics that are associated with the operations. For example, in a system that provides the Unset and Test&Set instructions, a write due to an Unset is a release that could be paired with a read due to a Test&Set which is an acquire. However, the write due to a Test&Set is not a release since it is not meant to be used to communicate the completion of previous memory operations of its processor.

Memory operations initiated by different processors are ordered by $\xrightarrow{hb1}$ only if paired release and acquire operations occur between them. For a more formal definition, we use the program order ( $\xrightarrow{po}$ ) relation, and the synchronization-order-1 ( $\xrightarrow{so1}$ ) relation defined on the synchronization operations of a sequentially consistent execution as follows.

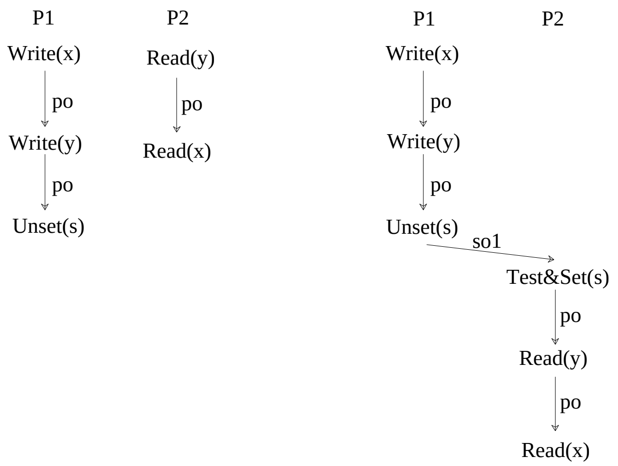
**Definition 2.2:** $s_1 \xrightarrow{so1} s_2$ iff $s_1$ is a release operation and $s_2$ is a paired acquire operation.

**Definition 2.3:** The $\xrightarrow{hb1}$ relation for a sequentially consistent execution is defined as ( $\xrightarrow{po} \cup \xrightarrow{so1}$ )$^+$, the irreflexive transitive closure of $\xrightarrow{po}$ and $\xrightarrow{so1}$.

The definitions of a data race, a data-race-free execution and a data-race-free program follow.

**Definition 2.4:** Two memory operations, *x* and *y*, in a sequentially consistent execution form a *race* $\langle x,y \rangle$, iff *x* and *y* conflict, and they are not ordered by the $\xrightarrow{hb1}$ relation of the execution. The race $\langle x,y \rangle$, is a *data race* iff at least one of *x* or *y* is a data operation. A sequentially consistent execution is *data-race-free* iff none of its operations form a data race. A program is *data-race-free* iff all its sequentially consistent executions are data-race-free.

Figures 1a and 1b respectively show sequentially consistent executions with and without data races. The *Pi's* denote processors, *op(x)* denotes a memory operation on the location *x*. Operations denoted by Read and Write are data operations and the remaining are synchronization operations. A write operation due to an Unset can be paired with a read operation due to a Test&Set. The execution of Figure 1a is not data-race-free since the conflicting write and read data operations of P1 and P2 respectively are not ordered by $\xrightarrow{hb1}$. The execution of Figure 1(b) is data-race-free because all conflicting data operations are ordered by $\xrightarrow{hb1}$, and no synchronization operation conflicts with a data operation.




(a) (b)

**Figure 1. Executions (a) with and (b) without data races.**

## 2.2. Weak Memory Models

We now briefly review and compare the weak memory models that we consider in this paper. For a data-race-free program, between two conflicting data operations of different processors there are explicit, hardware recognizable synchronization operations. This allows most of the actions to ensure sequential consistency to be delayed from the data operation to the subsequent synchronization operation, leading to higher performance. All four weak models considered in this paper are based on this general principle. For example, consider the data-race-free execution in Figure 1b. A conventional implementation of sequential consistency would stall on every memory operation until its completion. Specifically, P1 would stall on each of its writes. However, the weak models only require that either the writes of P1 are complete before the Unset is issued or before the Test&Set of P2 succeeds. Either approach guarantees sequential consistency for the execution in Figure 1b. To see how sequential consistency may be violated in a weak system in the presence of data races, consider the execution in Figure 1a. In this execution, it is possible that the new value of y written by P1 is propagated to P2 before the new value of x. Thus, it is possible for P2 to read the new value for y but the old value for x, thereby violating sequential consistency.

We now discuss how the four weak models differ from each other. The model of WO [DSB86] is specified as a set of explicit conditions on hardware. It has been shown that these conditions ensure sequential consistency for all data-race-free programs [AdH90, AdH91]. The model of $RC_{sc}$ [GLL90] is also specified as a set of conditions on hardware. The main difference between the conditions of WO and $RC_{sc}$ is that the latter exploit the distinction between acquire and release synchronization operations. The hardware conditions of $RC_{sc}$ are accompanied by a formalization of the programs for which they ensure sequential consistency. Such programs have been called properly labeled with respect to sequential consistency, and are the same as data-race-free programs [AdH91]. There do exist some programs that exhibit certain restricted kinds of data races for which WO and $RC_{sc}$ hardware ensure sequential consistency [AdH90]. These programs, however, are few and are not formally characterized. Hence for all practical purposes, we assume that for sequential consistency to be guaranteed by these models, programs should be data-race-free. DRF0 [AdH90] and DRF1 [AdH91] are defined to include all hardware that guarantees sequential consistency to programs that are data-race-free. DRF0 differs from DRF1 in that it does not distinguish between acquire and release operations.

# 3. A Hardware Condition for Dynamically Detecting Data Races

In this section, we first describe the problems that could potentially limit the use of current dynamic data race detection techniques on weak systems. We then give a hardware condition that addresses these limitations, and explain why this condition is already obeyed by all proposed implementations of the weak systems.

## 3.1. Problems in Applying Dynamic Techniques to Weak Systems

Existing dynamic data race detection techniques for sequentially consistent systems [AlP87, ChM91, DiS90, HKM90, NeM90, NeM91] instrument the program to record information about the memory operations of its execution. This information allows the $\xrightarrow{hb1}$ relation to be constructed, thereby allowing the detection of data races. There are two approaches for recording and using this information. The *post-mortem* techniques generate trace files containing the order of all synchronization operations to the same location, and the memory locations accessed between two synchronization operations in a given process. These trace files are analyzed after the execution; the ordering of the synchronization operations allows the $\xrightarrow{so1}$ relation of the execution to be constructed from which the $\xrightarrow{hb1}$ relation can be constructed. The pairs of conflicting operations (at least one of which is data) that are not ordered by the $\xrightarrow{hb1}$ relation are reported as data races. The *on-the-fly* techniques do not produce explicit trace files, but buffer partial trace information in memory and detect data races as they occur during the execution.

The difficulty in applying the above techniques on weak systems is that the weak execution may not be sequentially consistent (if the program is not data-race-free). For such an execution, we first need to formalize the

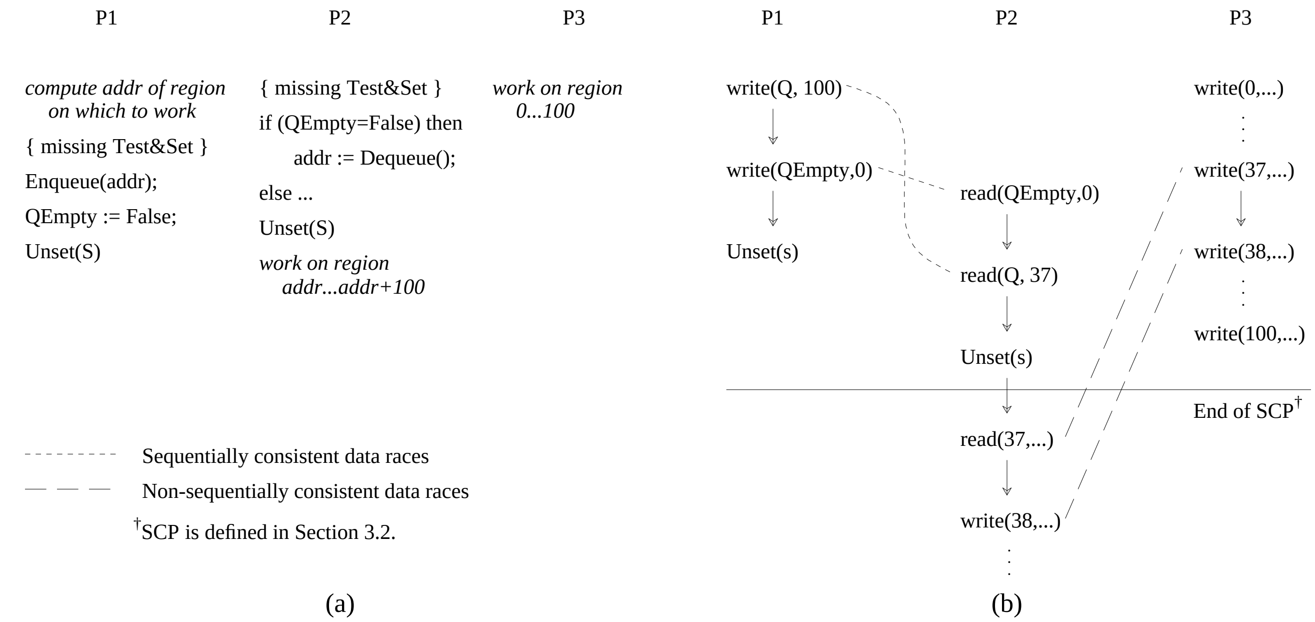



**Figure 2. (a) Program fragment, and (b) example happens-before-1 graph showing data races and SCP**

notion of a data race. This can be done in a manner analogous to that for sequentially consistent executions by defining the $\xrightarrow{so1}$ and $\xrightarrow{hb1}$ relations. Note that since in general, the synchronization operations of a weak system are not constrained to be executed in a sequentially consistent manner, the $\xrightarrow{so1}$ relation and hence the $\xrightarrow{hb1}$ relation may contain cycles and hence not be partial orders. Nevertheless, the current dynamic techniques for sequentially consistent executions can still be applied to find the data races of a weak execution.

Although data races of a weak execution can be easily detected with the current techniques, there are two potential problems that need to be resolved for the reported data races to be meaningful. The first problem is that on arbitrary weak hardware, it is theoretically possible for an execution to not exhibit data races and yet not be sequentially consistent. Fortunately, as we shall see later, the implementations of the weak models proposed to date do not exhibit this problem.

The second problem is that data races that may be reported from weak executions may never occur on any sequentially consistent execution of the program. This could easily occur on the current weak systems. Figure 2 illustrates a program fragment and one weak execution in which such non-sequentially consistent data races occur. In this program, processor P1 places the starting address of a region for processor P2 to work on in a shared queue and resets the QEmpty flag. Processor P2 checks to determine if work is available, and dequeues an address if the queue is not empty. Processor P3 works independently on some part of the address space. Since P1 and P2 both access the shared queue, operations manipulating the queue are enclosed in critical sections, implemented with the Test&Set and Unset instructions. However, due to an oversight, the Test&Set instructions were omitted. The program is therefore not data-race-free. A sequentially consistent execution of this program will exhibit data races between the accesses to the queue and the variable QEmpty. Because the program is not data-race-free, sequential consistency can be violated in an execution on a weak system. One such execution is shown in Figure 2b, where $op(x,a)$ represents a read or a write memory operation to location $x$ that respectively returns or stores the value $a$. In this execution, although processor P2 finds QEmpty to be reset, it does not read the new value, 100, enqueued by P1. Instead it reads an old value, in this case 37. The region that processor P2 starts working on now overlaps with the region accessed by P3. This gives rise to many data races between the operations of P2 and P3 as shown. On a sequentially consistent system, P2 could never have returned the value 37, and hence these races would never have occurred. Nevertheless, naively using the dynamic techniques would report all of these data races.

For debugging programs for weak memory systems, we are only interested in detecting data races that would also occur in a sequentially consistent execution of the program. Further, the motivation of detecting data races was to allow programmers to reason in terms of sequential consistency. Therefore, reporting data races that cannot occur in a sequentially consistent execution can be confusing to the programmer and can complicate the task of debugging. In the next sub-section, we develop a hardware condition that addresses this problem.

### 3.2. A Hardware Condition for Dynamic Data Race Detection

We develop a hardware condition such that for any execution on a system that obeys this condition, a set of data races that would also occur in some sequentially consistent execution can be identified. In Section 4, we will show how dynamic detection can be used to report this set.

Intuitively, our condition requires that an implementation guarantee sequential consistency until a data race actually occurs in the execution. Further, once a data race occurs, sequential consistency should be violated only in parts of the execution that are affected by the data race. This will ensure that every execution has a sequentially consistent prefix that contains the first data races (those not affected by others) of the execution. Dynamic techniques can then potentially be applied to this sequentially consistent prefix to report the first races. We introduce the following terminology to formalize our condition.

**Definition 3.1:** A *prefix* of an execution $E$ is a subset of the memory operations of $E$ such that if $y$ is in the prefix and $x \xrightarrow{hb1} y$ in $E$, then $x$ is in the prefix.

**Definition 3.2:** A prefix of an execution $E$ of a program $P$ is a *sequentially consistent prefix* or *SCP* of $E$ iff

(1) it is also the prefix of a sequentially consistent execution $E_{seq}$ of program $P$, and

(2) if $x$ and $y$ are in the prefix, then $\langle x,y \rangle$ is a data race in $E$ iff $\langle x,y \rangle$ is also a data race in $E_{seq}$.

An example of an SCP is shown in Figure 2b. Thus, for an execution $E$ of any program, the operations of a processor in its SCP are also the initial operations of the processor in some sequentially consistent execution, $E_{seq}$, of the program. Further, a data race involving operations in the SCP occurs in $E$ iff it also occurs in $E_{seq}$. This implies that the set of data races that have their operations in a particular SCP is a valid set of sequentially consistent data races to report.

To enable the identification of data races in an SCP, we propose a condition that in an execution, either a data race has its operations in a specific SCP of the execution, or the data race is affected by another data race with operations in the SCP, where ''affected'' is defined as follows.

**Definition 3.3:** A race $\langle x,y \rangle$ *affects* a memory operation $z$, written $\langle x,y \rangle \xrightarrow{A} z$, iff

(1) $z$ is the same memory operation as $x$ or $y$, or

(2) $x \xrightarrow{hb1} z$, or $y \xrightarrow{hb1} z$, or

(3) there exists a race $\langle x',y' \rangle$ such that $\langle x',y' \rangle \xrightarrow{A} z$, and either $\langle x,y \rangle \xrightarrow{A} x'$ or $\langle x,y \rangle \xrightarrow{A} y'$.

A race $\langle x,y \rangle$ affects another race $\langle x',y' \rangle$, written $\langle x,y \rangle \xrightarrow{A} \langle x',y' \rangle$, iff $\langle x,y \rangle \xrightarrow{A} x'$ or $\langle x,y \rangle \xrightarrow{A} y'$.

This implies that data races that are not affected by any other data race (intuitively the first data races) should always be in an SCP of the execution, i.e., they should also occur in a sequentially consistent execution. Thus, the data races that are not affected by any others constitute a valid set of data races that can be reported.

The formal hardware condition that we propose is as follows. We say that a race $\langle x,y \rangle$ occurs in an SCP if the operations $x$ and $y$ are in the SCP.

**Condition 3.4:** For any execution $E$ of a program $P$,

(1) if there are no data races in $E$, then $E$ is a sequentially consistent execution of program $P$, and

(2) there exists an SCP of $E$ such that a data race in $E$ either occurs in the SCP, or is affected by another data race that occurs in the SCP.

Condition 3.4(1) ensures that if no data races are reported, then the programmer can safely assume that the hardware is sequentially consistent, overcoming the first problem cited in the previous section. Condition 3.4(2) ensures that if a data race is detected in $E$, then there is a data race in $E$ that affects this data race that also occurs in some sequentially consistent execution of the program. Thus, the set of data races that are not affected by any other data race in $E$ form a valid reportable set of data races that also occur in some sequentially consistent execution.

### 3.3. Weak Hardware Already Obeys Condition for Dynamic Data Race Detection

In this subsection, we give the most significant result of our paper.

**Theorem 3.5**: Condition 3.4 for dynamic debugging is already obeyed by all implementations of WO and $RC_{sc}$ and all proposed implementations of DRF0 and DRF1.

Due to space constraints, in this paper, we only give an informal intuition for this result. A formal proof of the theorem appears in [AHM91]. For brevity, we use the term *all weak implementations* to imply all possible implementations of WO and $RC_{sc}$ and all implementations of DRF0 and DRF1 that have been proposed to date.

Condition 3.4(1) requires that for a data-race-free execution on a weak system, the weak hardware should appear sequentially consistent. All weak implementations guarantee sequential consistency to data-race-free programs. They achieve this by guaranteeing sequential consistency to every data-race-free execution on the weak system [AdH90, AdH91, GLL90], thereby obeying Condition 3.4(1).

Condition 3.4(2) requires that every data race in a weak execution that is not affected by any other data race should also occur in a specific sequentially consistent execution of the same program. The data races that are not affected by any others are intuitively, the first data races in an execution. Therefore, Condition 3.4(2) can be obeyed by ensuring that an execution provides sequential consistency until a data race actually occurs. Even then, a violation of sequential consistency should be allowed to occur only for operations that are directly affected by the data race. Intuitively, this is true for all weak implementations for the following reason. Weak implementations are allowed to violate sequential consistency only for executions that exhibit data races. Practically, however, we expect that it is not possible to predict whether an execution will exhibit a data race until a data race actually occurs in the execution. Thus, all weak implementations provide sequential consistency until a data race actually occurs in the execution, and violate sequential consistency only in the parts of the execution that are affected by the data races. The formal proof for this condition shows the existence of an SCP for every execution on a weak implementation such that either all operations of a processor are in the SCP, or the first operation of a processor not in the SCP is affected by a data race in the SCP.

## 4. Detecting Data Races on Weak Hardware

In this section we show how Condition 3.4 can be used to dynamically detect sequentially consistent data races on weak systems. We employ a post-mortem approach to locate sets of data races, each of which contains at least one that belongs to a specific SCP. Recall that a data race occurs in an SCP only if it also occurs in the corresponding sequentially consistent execution. Our approach thus directs the programmer to data races that would have also occurred in some sequentially consistent execution of the program. In Section 5 we argue that any limitations of this technique are analogous to those that occur with dynamic detection on sequentially consistent hardware. We also explore the possibility of using an on-the-fly approach in Section 5.

### 4.1. Program Instrumentation

So far, data races and hardware constraints have been expressed in terms of individual operations. Detecting data races at such a low level would require tracing the program order of all memory operations performed by each processor, which in general would be impractical. Instead, higher-level information about the execution can be recorded and used for data race detection. The execution of each processor can be viewed as a sequence of *events* that represent groups of memory operations. We adopt the approach used by previous data race detection methods [AlP87, ChM91, DiS90, HKM90, NeM90, NeM91] and define an event to be either a single synchronization operation (a *synchronization event*), or a group of consecutively executed data operations (a *computation event*). For example, the events of the execution in Figure 2b are shown in Figure 3 as sets of memory operations enclosed within rectangles. Each event, $A$, has two attributes, $READ(A)$ and $WRITE(A)$, the sets of memory locations read and written by the event (upper case letters denote events). Such a higher-level view is useful since recording the *READ* and *WRITE* sets is in general more efficient than tracing every memory operation performed by the event. For example, bit-vectors representing those (shared) variables that might be accessed between two synchronization events can be constructed, and when a variable is accessed, the corresponding bit is set. The bit-vector is then written to a trace file at the end of each computation event. Such an approach avoids writing a trace record for every memory operation.

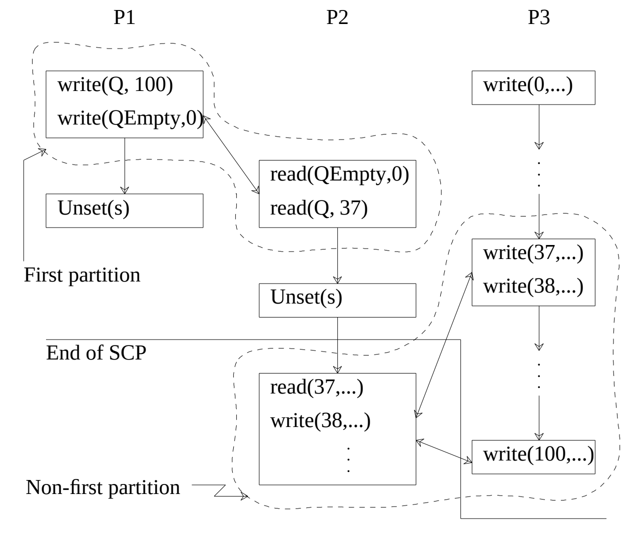



**Figure 3. Augmented happens-before-1 graph showing first and non-first data race partitions**

We assume that the program is instrumented to produce trace files containing (1) the execution order of events issued by the same processor, (2) the relative execution order of synchronization events involving the same location, and (3) the *READ* and *WRITE* sets for each computation event. From (1) and (2), the $\xrightarrow{hb1}$ relation (defined over events, instead of operations) can be represented by constructing the *happens-before-1 graph*, which contains one node for every event, and edges between nodes to represent the $\xrightarrow{po}$ and $\xrightarrow{so1}$ relations. Data races are defined for computation events in the same way as they were defined for data operations: a data race between two computation events, $\langle A,B \rangle$, exists if $A$ and $B$ access a common location that at least one of them writes, and no path connects $A$ and $B$ in the happens-before-1 graph. We use the term *lower-level data race* to refer to a data race among memory operations (e.g., $\langle a,b \rangle$), and the term *higher-level data race* (or just *data race*) to refer to a data race among events (e.g., $\langle A,B \rangle$). Since $A$ and $B$ represent groups of memory operations, the data race $\langle A,B \rangle$ may represent many lower-level data races among the individual operations comprising $A$ and $B$. If at least one of these lower-level data races belongs to an SCP, then we say that the higher-level data race does also.

### 4.2. Locating Data Races in an SCP

We now show how to locate data races that are in an SCP. Intuitively, we try to identify the first data races that occurred.

We add edges to the happens-before-1 graph, $G$, to construct an augmented graph, $G'$. The purpose of these edges is to capture the possible effect one data race may have on another. For each race, we add a doubly directed edge between the two nodes involved in the race. Then, given two races, $\langle A,B \rangle$ and $\langle C,D \rangle$, a path exists in $G'$ from $A$ (or $B$) to $C$ (or $D$) iff $\langle A,B \rangle \xrightarrow{A} \langle C,D \rangle$. The graph $G'$ for the execution in Figure 2b is shown in Figure 3.

If for an execution, the augmented graph $G'$ were acyclic, it would define a partial ordering among the data races. By construction, the first data races as given by the partial order would not be affected by any other races, and hence by Condition 3.4(2) these races would be guaranteed to belong to an SCP. However, since $G'$ may contain cycles, we need to partition the data races and define a partial ordering among the partitions. The partitions will be defined so that at least one data race in each partition is guaranteed to belong to an SCP.

We use the strongly connected components[1] of $G'$ to partition the data races; two data races belong to the same partition if and only if the events involved in the races belong to the same strongly connected component. The first partitions are then identified by defining a partial ordering, $\xrightarrow{P}$, on the partitions as follows:

**Definition 4.1:** $Part_1 \xrightarrow{P} Part_2$ iff a path exists in $G'$ from some event in $Part_1$ to some event in $Part_2$.

1. A strongly connected component of a directed graph has the property that paths exist between each pair of nodes in the component, but no paths exist from a node in one component to a node in another component and back.

A partition is *first* if it is not ordered by $\xrightarrow{P}$ after any other partitions containing at least one data race.

For the execution of Figure 2b, the partitions and their ordering are shown in Figure 3. We now state the following theorems regarding the first partitions. The proofs of these theorems are relatively straightforward and appear in [AHM91].

**Theorem 4.1:** There are no first partitions containing data races iff no data races were exhibited in the execution.

**Theorem 4.2:** In each first partition containing data races, at least one data race belongs to an SCP.

We now claim that for an execution on a weak system, only the first partitions containing data races should be reported to the programmer. This is sufficient because of the following reason. If for some execution, no first partitions are reported, then by Theorem 4.1, no data races must have occurred in the execution and by Condition 3.4(1), this execution must be sequentially consistent. If first partitions are reported, then by Theorem 4.2, at least one of the data races in each of these partitions would occur on a sequentially consistent execution of the program.

A limitation of our method is that for first partitions with more than one data race, we cannot detect precisely which race is sequentially consistent. Nevertheless, the portion of the execution that has to be examined by the programmer to detect the sequentially consistent data race is significantly narrowed, making debugging much easier. In the next section, we discuss how dynamic race detection techniques in sequentially consistent systems suffer from an analogous limitation.

## 5. Data Race Detection on Sequentially Consistent Systems vs. Weak Systems

The limitations of our approach for detecting data races on weak systems are analogous to those for detecting data races on sequentially consistent systems. We next discuss these analogies, focusing on accuracy and overhead.

The effectiveness of data race detection depends on the accuracy with which the reported data races reflect program bugs. Our method reports partitions of data races, where each partition contains at least one sequentially consistent or SCP data race, i.e., a true program bug. However, the method suffers from the following limitations: (1) we cannot always pinpoint exactly which data races in each first partition belong to the SCP and hence are the valid races, and (2) we do not locate data races in non-first partitions that may also belong to the SCP.

Dynamic techniques for sequentially consistent systems also suffer from limitations analogous to the above.

There is an analogy between data races in weak systems that are not in an SCP and data races in sequentially consistent systems that are artifacts of other data races. A data race is an artifact if it occurs only because a previous data race left the program's data in an inconsistent state unexpected by the programmer, and hence is not a direct manifestation of a program bug. Methods for accurately locating data races on sequentially consistent systems [NeM90, NeM91] also order partitions of data races to enable detection of the non-artifact races. These methods also suffer from the above limitations: (1) they identify partitions of data races, each of which is guaranteed to contain at least one non-artifact data race, but cannot determine precisely which races within each partition are not artifacts [NeM91], and (2) they cannot identify non-artifact data races that happen to belong to non-first partitions. When using the first partitions, we expect that the difference between debugging on weak or sequentially consistent systems will be insignificant; we believe reasoning about an artifact data race on a sequentially consistent system is as difficult as reasoning about non-SCP data races on weak systems. Nevertheless, for both the sequentially consistent and weak systems, identifying the first partitions significantly narrows the portion of the execution that must be investigated to locate the non-artifact or the SCP data races.

Another issue impacting accuracy is the reliability of the trace data. Again, sequentially consistent systems and weak systems share common limitations. Program instrumentation can be added by a trusted facility (such as a compiler) and so can generally be assumed correct. However, pathological programs can be constructed that behave unpredictably after a data race (because the program's data is left in an inconsistent state) and randomly overwrite the program's own address space. In the worst case, the traces can be overwritten with erroneous data suggesting that no data races were exhibited, when in fact data races occurred. Such pathological cases can be constructed for both weak and sequentially consistent systems; the reliability problem is no worse in weak systems. In practice, however, we expect trace corruption to rarely occur.

Another important factor influencing the effectiveness of data race detection is overhead. Section 4 presented a post-mortem approach in which trace files are written during execution and analyzed during a post-mortem phase. Overhead is incurred both during execution and during post-mortem analysis. However, the overhead of our method is no worse than post-mortem methods on sequentially consistent systems: we require no more execution-time information that these methods, and our analysis requires computation similar to the more accurate techniques for sequentially consistent systems [NeM90, NeM91].

Another approach for locating data races is *on-the-fly* detection. On-the-fly approaches have the advantage of not consuming secondary storage [ChM91, DiS90, HKM90], but existing methods are typically less accurate and have higher run-time overhead than post-mortem techniques. The loss of accuracy is a result of attempts to keep space overhead low by only buffering limited trace information in memory. As a result, some of the first data races can remain undetected. This is a problem for both sequentially consistent systems (where non-first data races can be artifacts) and weak systems (where non-first data races may not belong to an SCP). Future work includes investigating how our method might be employed on-the-fly to locate the first data races.

We believe that once data races are detected using our methods, other debugging tools for sequentially consistent systems can be effectively applied on weak systems as well. This is because the part of the execution that contains the ‘‘first’’ bugs is sequentially consistent and can be debugged as on a sequentially consistent execution. Since the overhead and accuracy of our data race detection method on weak systems are similar to that on sequentially consistent systems, we expect that debugging in general on weak systems will also be no more difficult. Programs for weak systems can therefore be debugged while taking advantage of the performance gains achieved by these systems; a slower sequentially consistent mode for debugging is not necessary.

## 6. Conclusions

The shared memory models of weak ordering, release consistency with sequentially consistent synchronization operations, data-race-free-0 and data-race-free-1 provide high performance by guaranteeing sequential consistency mainly to programs that do not exhibit data races on sequentially consistent hardware. To allow programmers to use the intuition and algorithms already developed for sequentially consistent systems, it is important to determine when a program is data-race-free, and when it is not, to identify the parts where data races could occur.

Detecting data races is also crucial for programs written for sequentially consistent systems. Static techniques for sequentially consistent systems can be directly applied to weak systems as well. Dynamic techniques, on the other hand, may report data races that could never occur on sequentially consistent systems. This can complicate debugging because programmers can no longer assume the model of sequential consistency.

We have shown how a post-mortem dynamic approach can be used to detect data races effectively even on weak systems. The key observation we make is that in practice weak systems preserve sequential consistency at least until the first data races (those not affected by any others) since they cannot predict if a data race will occur. We formalize this condition by using the notion of a sequentially consistent prefix. For an execution on a system that obeys this condition, we show how we can either (1) correctly report no data races and conclude the execution to be sequentially consistent or (2) report the first data races that also occur on a sequentially consistent execution (within the limitation discussed in Section 4.2). Since our condition is already met by all implementations of WO and $RC_{sc}$ and all proposed implementations of DRF0 and DRF1, our technique can exploit the full performance of the weak systems. Further, we have shown that any limitations of our technique are also shared by dynamic techniques on sequentially consistent systems. Finally, we believe that the presence of the sequentially consistent prefix also allows other debugging tools of sequentially consistent systems to be used on weak systems.

We have demonstrated the use of our hardware condition with a post-mortem dynamic technique. Using on-the-fly techniques for weak systems has problems analogous to those on sequentially consistent systems and is a subject of future work.

Although the definitions and theorems presented in this paper may be considered complex, only those who wish to verify our results must confront this complexity. Designers of memory systems can simply check that their hardware obeys Condition 3.4, while users of our data race detection technique can reason about their program as if the hardware were sequentially consistent.

**References**

[AdH90] S. V. ADVE and M. D. HILL, Weak Ordering - A New Definition, *Proc. 17th Annual Intl. Symp. on Computer Architecture*, May 1990, 2-14.

[AHM91] S. V. ADVE, M. D. HILL, B. P. MILLER and R. H. B. NETZER, Detecting Data Races on Weak Memory Systems, Computer Sciences Technical Report, To be Published, University of Wisconsin, Madison, 1991.

[AdH91] S. V. ADVE and M. D. HILL, An Approach for Specifying Shared Memory Models, Computer Sciences Technical Report, To be Published, University of Wisconsin, Madison, 1991.

[AlP87] T. R. ALLEN and D. A. PADUA, Debugging Fortran on a Shared Memory Machine, *Proc. Intl. Conf. on Parallel Processing*, August 1987, 721-727.

[BaK89] V. BALASUNDARAM and K. KENNEDY, Compile-time Detection of Race Conditions in a Parallel Program, *3rd Intl. Conf. on Supercomputing*, June 1989, 175-185.

[Ber66] A. J. BERNSTEIN, Analysis of Programs for Parallel Processing, *IEEE Trans. on Electronic Computers EC-15*, 5 (October 1966), 757-763.

[ChM91] J. CHOI and S. L. MIN, Race Frontier: Reproducing Data Races in Parallel Program Debugging, *Proc. 3rd ACM Symp. on Principles and Practice of Parallel Programming*, April 1991.

[DiS90] A. DINNING and E. SCHONBERG, An Empirical Comparison of Monitoring Algorithms for Access Anomaly Detection, *Proc. ACM SIGPLAN Notices Symp. on Principles and Practice of Parallel Programming*, March 1990, 1-10.

[DSB86] 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.

[EmP88] P. A. EMRATH and D. A. PADUA, Automatic Detection of Nondeterminacy in Parallel Programs , *Proc. SIGOPS Workshop on Parallel and Distributed Debugging*, May 1988, 89-99. Also appears in *SIGPLAN Notices* **24**(1) (January 1989).

[GLL90] 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.

[Goo91] J. R. GOODMAN, Cache Consistency and Sequential Consistency, Computer Sciences Technical Report #1006, Univ. of Wisconsin, Madison, February 1991.

[HKM90] R. HOOD, K. KENNEDY and J. MELLOR-CRUMMEY, Parallel Program Debugging with On-the-fly Anomaly Detection, *Supercomputing '90*, November 1990, 74-81.

[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.

[NeM90] R. H. B. NETZER and B. P. MILLER, Detecting Data Races in Parallel Program Executions, *To appear in Research Monographs in Parallel and Distributed Computing, MIT Press, 1991.* Also available as *Proc. 3rd Workshop on Programming Languages and Compilers for Parallel Computing, August 1990.*

[NeM91] R. H. B. NETZER and B. P. MILLER, Improving the Accuracy of Data Race Detection, *Proc. 3rd ACM Symp. on Principles and Practice of Parallel Programming*, April 1991.

[Tay83a] R. N. TAYLOR, A General-Purpose Algorithm for Analyzing Concurrent Programs, *Communications of the ACM 26*, 5 (May 1983), 362-376.

[Tay83b] R. N. TAYLOR, Complexity of Analyzing the Synchronization Structure of Concurrent Programs, *Acta Informatica 19*(1983), 57-84.