SHARED MEMORY CONSISTENCY CONDITIONS FOR NON-SEQUENTIAL EXECUTION: DEFINITIONS AND PROGRAMMING STRATEGIES*

HAGIT ATTIYA†, SOMA CHAUDHURI‡, ROY FRIEDMAN§, AND JENNIFER L.WELCH¶

Abstract.

To enhance performance on shared memory multiprocessors, various techniques have been proposed to reduce the latency of memory accesses, including pipelining of accesses, out-of-order execution of accesses, and branch prediction with speculative execution. These optimizations can, however, complicate the user's model of memory. This paper attacks the problem of simplifying programming on two fronts.

First, a general framework is presented for defining shared memory consistency conditions that allows non-sequential execution of memory accesses. The interface at which conditions are defined is between the program and the system, and is architecture-independent. The framework is used to generalize three consistency conditions—sequential consistency, hybrid consistency, and weak consistency—for non-sequential execution. Thus familiar consistency conditions can be precisely specified even in optimized architectures.

Second, several techniques are described for structuring programs so that a shared memory that provides the weaker (and more efficient) condition of hybrid consistency appears to guarantee the stronger (and more costly) condition of sequential consistency. The benefit is that sequentially consistent executions are easier to reason about. The first technique statically classifies accesses based on their type. This approach is extremely simple to use and leads to a general methodology for writing efficient synchronization code. The second technique is to avoid data races in the program, which was previously studied in a somewhat different setting.

Precise, yet short and comprehensible, proofs are provided for the correctness of the programming techniques. Such proofs shed light on the reasons these techniques work; we believe that the insight gained can lead to the development of other techniques.

Key words. Distributed Shared Memory, Consistency Conditions, Sequential Consistency

AMS subject classifications. [[What goes in here?]]

PII. [[What goes in here?]]

* An extended abstract of this paper appeared in *Proc. of the 5th ACM Symposium on Parallel Algorithms and Architectures,* June/July 1993, pp. 241–250. This work was supported by grant No. 92-0233 from the United StatesS-Israel Binational Science Foundation (BSF), Jerusalem, Israel, Technion V.P.R.—Argentinian Research Fund, the fund for the promotion of research in the Technion, NSF grants CCR-89-15206, CCR-9010730, and CCR-9308103, DARPA contracts N00014-89-J-1988 and N00014-92-J-4033, ONR contract N00014-91-J-1046, a grant from the ISU Graduate College, an IBM Faculty Development Award, NSF Presidential Young Investigator Award CCR-9158478, and TAMU Engineering Excellence funds.

http://www.siam.org/journals/

† Department of Computer Science, The Technion, Haifa 32000, Israel. Email: `hagit@cs.technion.ac.il`

‡ Department of Computer Science, Iowa State University, Ames, IA 50011. Email: `chaudhur@cs.iastate.edu`. Much of this work was performed while this author was with the Laboratory for Computer Science, Massachusetts Institute of Technology Cambridge, MA 02139.

§ Department of Computer Science, Cornell University, Ithaca, NY 14853, Email: `roy@cs.cornell.edu`. This work was performed while the author was with the Department of Computer Science, The Technion, Haifa 32000, Israel.

¶ Department of Computer Science, Texas A&M University, College Station, TX 77843-3112. Email: `welch@cs.tamu.edu`. Much of this work was performed while this author was with the Department of Computer Science, University of North Carolina, Chapel Hill, NC 27599-3175.

1. Introduction.

1.1. Overview. Shared memory multiprocessors are an interesting alternative to message-passing distributed computer systems. The parallelism in multiprocessors offers the potential of greatly increased performance, and shared memory is an attractive communication paradigm. Unfortunately, access to shared memory locations is a major bottleneck for the performance of multiprocessors. The high latency of memory operations is due to the time to execute memory operations locally, and the inter-processor communication delay, which increases with the number of processors.

Many uniprocessor architecture techniques have been developed and implemented to hide the latency of memory operations by allowing operations to overlap. These techniques include performing memory accesses in parallel, pipelining memory accesses, initiating accesses out of order, and speculative execution (in which the system predicts the outcome of future conditional branches and begins memory accesses for the predicted branch). Specific instances include [1, 20, 24, 34, 35, 37, 39, 40]. Since all these techniques deviate from the sequential order of memory accesses specified by the program, we call them "non-sequential".

In multiprocessors, memory is often replicated and distributed to compensate for the inter-processor communication delay. A consistency condition for shared memory specifies what guarantees are provided about the values returned in the presence of concurrent accesses to different copies of the memory. A variety of consistency conditions have previously been proposed for shared memory architectures, e.g., [3, 4, 8, 9, 12, 18, 19, 21, 23, 25, 28, 29]. Clearly, the consistency condition limits the optimizations that can be implemented.

A natural question is how to define consistency conditions that allow non-sequential execution of memory accesses, in a manner that admits optimized systems. A further question is how to program these optimized systems with these generalized consistency conditions in a safe and effective manner.

In this paper, we focus on the interface between programs and the shared memory under consistency conditions that allow non-sequential execution. We present a framework for defining such conditions as properties of executions, rather than as properties of hardware implementations. We hope that this aspect of our framework will elucidate these somewhat complex conditions for algorithm designers, programmers, and perhaps compiler designers. We demonstrate our framework by extending three consistency conditions to allow for non-sequential execution of memory accesses. For one particular condition that allows efficient implementations, we present and prove correct several programming techniques that provide the illusion of a stronger and more natural condition. The first approach is based on a static labeling of the memory accesses in the program according to their type. The second approach is to access shared data only inside appropriately protected critical sections. The third approach is to avoid data races, an approach pioneered in [3, 4, 22, 23] for hardware implementations.

1.2. Detailed Description. Our first contribution is a framework for defining consistency conditions that is general enough to allow non-sequential execution of memory operations. This framework combines the following two features: (1) The interface at which conditions are specified in our model is between the program and the system instead of specifying the internal behavior of the system. This is the interface

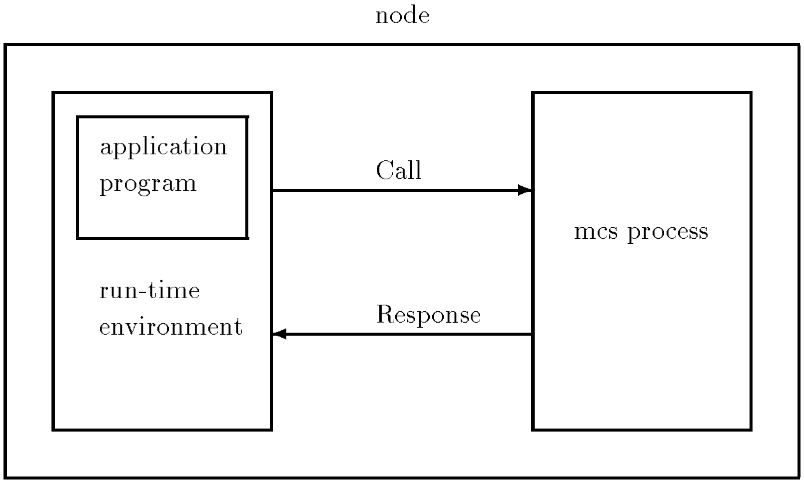

FIG. 1. *A node.*

used in [12, 25, 28] and in our opinion is the natural one to use for specification to be independent of implementation. (2) The framework allows for arbitrary optimizations by the system, including especially speculative execution of memory accesses. Recent experiments have shown that if the program has complex control flow then only moderate speedup can be achieved by parallelism without speculative execution (e.g., [26, 41]).

The framework is then used to extend three known consistency conditions for non-sequential execution. Our extensions have two pleasing properties: (1) The conditions are defined for all programs, not just programs that satisfy certain conditions. (2) We give a formal, yet intuitive, treatment of explicit control instructions, which are crucial for expressing the flow of control in a program and in analyzing its correctness on non-sequential implementations.

Our framework assumes a system consisting of a collection of nodes. At each node there is an *application program*, a *memory consistency system* (mcs) process, and a *run-time environment*. An illustration of a node is given in Figure 1. An application program contains instructions to access shared memory and conditional branch instructions. The mcs processes at all the nodes collectively implement the shared objects that are manipulated by the application programs. The run-time environment at a node executes the shared memory instructions by interacting with the local mcs process; its decisions as to which instructions to execute rely on the application program at that node. (We use the term run-time environment to refer to the combination of the functionality of a compiler, which sees the whole program, and a conventional run-time system, which makes decisions dynamically based on partial knowledge.) In our framework, consistency conditions are specified as a guarantee on the run-time environments at all nodes with respect to the program at each node. It is the responsibility of both the mcs process and the run-time environment to provide this guarantee.

A straightforward run-time environment would simply submit operations to the mcs one at a time in the order specified by the program. In order to achieve various optimizations however, the run-time environment might submit operations out of order, might have multiple operations pending at a time, and might anticipate branches (sometimes incorrectly). We do not address the specific algorithm used by the run-time environment. (That is another very interesting problem, beyond the scope of this paper.) Instead, our goal is to model the run-time environment sufficiently abstractly so that any of a large number of specific run-time environments can fit into this framework. Obviously the run-time environment cannot do just anything—the optimizations that it performs should be transparent to the application program. The strongest condition we require of a correct run-time environment is that there exist a way (after the fact) to take a subset of the operations performed by the run-time environments at all the nodes and order them to be consistent with some "sequential execution" of the programs at all the nodes. (Some of the operations performed by the run-time environment might end up not being used, for instance if they resulted from an incorrect prediction about a branch. These operations can be ignored in determining whether there is a corresponding sequential execution.) We emphasize that the order in which operations *appear* to execute is what is important, not the order in which they *actually* execute.

Our framework incorporates rollback and compensating operations in an implicit manner. In particular, we allow the run-time environment to communicate with the mcs in order to perform other operations on the data that are not part of the application programs but that are necessary for achieving the desired consistency condition (for example, operations to restore the state of the shared variables due to incorrect predictions). These operations are ignored when the subset of operations consistent with a sequential execution is taken.

Given this framework, we generalize three known consistency conditions—sequential consistency, weak consistency and hybrid consistency.

Our second contribution addresses the issue of writing programs for hybrid consistency and formally proving their correctness. *Hybrid consistency* is an efficient and expressive consistency condition [12]; it captures some of the essential features of several other consistency conditions appearing in the literature [3, 15, 18, 19]. Memory access operations are classified as either *strong* or *weak*. This classification is crucial to the definition of the consistency condition, as observed by the programmer: A global ordering is imposed on strong operations at different processes, but little is guaranteed about the ordering of weak operations at different processes beyond what is implied by their interleaving with the strong operations. The classification also provides hints to the run-time environment concerning which optimizations can be applied to which accesses. Clever use of the classification can improve the performance of programs.

Unfortunately, it is more difficult to program memories that support hybrid consistency than to program memories that support sequential consistency, since the guarantees provided by the former are weaker than those provided by the latter. For instance, how should the programmer decide which accesses should be strong and which should be weak in order to improve performance yet still ensure correctness? A way to cope with this problem is to develop rules and transformations for executing programs that were written for sequentially consistent memories on hybrid consistent memories. The benefit is that sequentially consistent executions are easier to reason about while hybrid consistency can be implemented more efficiently. We consider several techniques for turning programs written for sequential consistency into programs that work for hybrid consistency. Precise, yet short and comprehensible proofs are provided for the correctness of the programming techniques. Such proofs shed light on the reasons these techniques work; we believe that the insight gained can lead to the development of other techniques.

The first approach we present is based on statically labeling specific accesses as strong, according to their type. We prove that programs in which all writes are strong run on hybrid consistent shared memory implementations as if they were sequentially consistent. There is a symmetric result that every hybrid consistent execution of a program in which all reads are strong is sequentially consistent [11]. This theorem requires further assumptions on the execution; although these assumption are shown to be necessary, they make the strong read technique impractical.

The second approach is based on the mutual exclusion paradigm and uses the first approach as a tool. One proposed way to program with hybrid-like consistency conditions [18, 21] is to protect accesses to shared data with critical sections and then to use strong operations in the mutual exclusion code and weak operations inside the critical section. When the critical sections are significant in size, the extra cost to execute the strong operations in the mutual exclusion algorithm is more than compensated for by the efficiency of the weak operations in the critical sections. We take a careful look at this paradigm for hybrid consistency and show that it is applicable, although care must be taken. Specifically, we show that many mutual exclusion algorithms designed to work on a sequentially consistent memory can be modified to work correctly on a hybrid consistent memory. The modification is to label all writes in the entry and exit sections as strong and all other operations as weak and to insert a dummy write in an appropriate place. However, this transformation only works for *non-cooperative* algorithms, in which processes do not participate in the mutual exclusion protocol unless they are actively vying for entry to the critical section.

The third approach for programming with hybrid consistency is to run data-race-free programs. This approach was pioneered by [3, 4, 21, 22, 23] in the context of hardware implementations. (See Section 2 for a more detailed discussion.) A *data race* occurs when two accesses to the same location occur, at least one is a write, and there is no synchronization between them. Data races in a program are considered bad practice: They add to the uncertainty of concurrent programs, beyond what is already implied by the fact that different processes may run at different rates and memory accesses may have variable duration. (Some debuggers even regard data races as bugs in the program.) Methods have been developed to detect and report data races, also called *access anomalies*; e.g., [6, 16, 18, 17, 30, 32, 33]. It is reasonable to assume that data-race-free programs account for a substantial portion of all concurrent programs. We formally prove that data-race-free programs run on hybrid consistent shared memory implementations as if they were sequentially consistent; this result is shown in our programmer-oriented framework.

Although many parallel programs are expected to be data-race-free, we cannot ignore the drawbacks of these programs. Proving that a program is data-race-free, even for restricted cases, is co-NP-hard [31]. Also, it is sometimes difficult to find the exact location of the data race in the program [32]. Our static methods provide an alternative and show that it is not necessary to make a program data-race-free in order to guarantee correct behavior. These methods are especially well-suited to applications in which reads greatly outnumber writes (or vice versa).

The rest of the paper is organized as follows. Section 2 discusses related work.

Section 3 describes our system model. The modified definitions of sequential consistency, weak consistency and hybrid consistency are given in Section 4. The static methods for programming with hybrid consistency are discussed in Section 5. Data-race-free programs are discussed in Section 6. We conclude with a discussion of the results in Section 7.

2. Related Work. Many existing formal treatments of memory consistency conditions [12, 23, 25, 28, 38] assume that memory operations are executed sequentially—one at a time and in program order. Several recent papers proposed formalisms to allow some non-sequential optimizations [3, 4, 21, 22, 23]. In this section, we compare our work with these formalisms.

Gibbons, Merritt, and Gharachorloo developed a framework for defining consistency condition which is based on a series of I/O automata, and used this framework to define *release consistency*, a formalization of the Stanford DASH shared memory model [22, 23]. This definition, as well as the definition of weak ordering [18, 19], are given at the interface between the mcs and the network. Hence, the resulting consistency conditions are not convenient for programmers and theoreticians. In particular, the definitions in [22, 23] are very detailed and complex. In contrast, in our framework as well as in [12, 25, 28], consistency conditions are defined at the interface between the application program and the system; consistency conditions are defined by describing the way operations are viewed (or ordered) and not the way operations are executed. Hence, our approach is more programmer oriented.

Adve and Hill developed another programmer oriented formalism, called *SCNF* [2, 5], using the assumption that programmers always want to work assuming sequentially consistent memories. In this approach, a consistency condition is a contract between software and hardware. That is, hardware must behave as if it was sequentially consistent for all programs that obey certain properties; various consistency conditions differ in the properties they require from programs. Examples of such conditions include, e.g., DRF0 [3], DRF1 [4], PL_{pc1} and PL_{pc2} [2]. For each of these programming models, Adve and Hill introduce sufficient conditions for hardware; hardware that follows these conditions executes programs that obey the corresponding programming model as if it was sequentially consistent. The problem with this approach is that the resulting consistency conditions are not defined for programs that do not obey the required properties. This limits the programming style, even in cases where other programming styles could yield better performance. The sufficient conditions introduced by Adve and Hill for DRF0, DRF1, PL_{pc1}, and PL_{pc2} [2] extend the definition of these condition for all programs. The approach taken by Adve and Hill is to develop a programming model, and then suggest how to tailor the hardware to support it. We take the complementary approach, first developing a consistency condition, and then developing programming methods for it.

Our framework includes an explicit model of the program, including control instructions. Our notion of the control flow of the program is derived syntactically from the code. Adve and Hill's sufficient conditions for DRF0, DRF1, PL_{pc1}, and PL_{pc2} [2], which permits non-sequential execution of memory operations, is based on the notion of a read operation controlling a write operation by the same processor. It is not obvious that this notion captures all the possible ways one operation can control another. Other previous work on specifying consistency conditions has either totally ignored control instructions [7, 12, 14, 25, 28] or has only made the informal requirement that uniprocessor control dependences are preserved [3, 21, 23].

Gibbons and Merritt [22] present a framework that deals formally with pipelining of memory operations. Their framework, recast in our terms, is based on the run-time environment submitting the memory accesses to the mcs, together with a partial order restricting the allowable reorderings of the accesses. Because the interaction between the run-time environment and the mcs is based on partial orders, their framework does not encompass arbitrary out-of-order or speculative execution of operations and it seems difficult to extend it to do so. We have taken a complementary approach and focused on modeling the program and its behavior, leaving unspecified the details of the interaction between the run-time environment and the mcs. Consequently, our framework allows arbitrary out-of-order and speculative execution of operations.

The idea that data-race-free programs can be executed on more relaxed implementations of shared memory as if they were sequentially consistent was pioneered in [2, 3, 4]. Similar results for other consistency conditions were also proved in [10, 22, 23]. We have applied this approach for hybrid consistency, using our framework. Admittedly, hybrid consistency may disallow some optimizations considered in several of the above papers. However, we believe the accessibility of our technical development provides insight concerning the interplay between conditions on the program and memory consistency conditions.

Another approach was taken by Shasha and Snir ([37]). They consider multiprocessor programs, written assuming sequential consistency, and investigate where to insert memory barrier operations (fences), so that if operations are executed in pipeline, but the fences are obeyed, then the result is as if the programs were executed sequentially. Their results are based on a partial order that have to hold between instruction instances, representing program order and causality. The execution is sequentially consistent only if this order is acyclic. Therefore, they insert the minimal amount of fences that will make the induced partial order acyclic.

Singh proved sufficient conditions for executions generated by optimized hardware, e.g., Pipelined RAM, causal memory and hybrid consistency, to be sequentially consistent [38]. Unlike the approach taken in this paper and by other researchers in this area, e.g., [2, 3, 4, 10, 22, 23], the approach taken by Singh examines the executions generated by the optimized hardware and not the programs that should be run on it.

The framework developed in this paper was extended in [13] to formally define *alpha consistency*, capturing the semantics provided by DEC-Alpha based multiprocessors (see Section 4.1). Two programming methodologies, similar to the ones developed in this paper, are presented in [13]: (a) every data-race free program runs on an alpha consistent memory as if it was sequentially consistent, and (b) any non-cooperative mutual exclusion algorithm based on sequential consistency can be transformed into a correct solution based on alpha consistency. However, since the Alpha does not support strong operations in the same sense as hybrid consistency (or release consistency),[1] the definition of data-race free programs and the method for handling mutual exclusion algorithms in [13] are different from those in this paper.

3. Framework. In this section, we present our formal definitions.

[1] The *memory-barrier* operation of the Alpha does not impose any interprocess ordering of operations.

3.1. System Components. An *application program* consists of a sequence of instructions, each with a unique label. There are two (disjoint) types of instructions, (shared) memory instructions and control instructions. A *memory instruction* specifies an access to a shared object. The specific kind of access depends on the data type of the object.[2] A *control instruction* consists of a *condition* (a boolean function of the process' local state) and a *branch* (jump to the instruction with the given label).

The *memory consistency system* (mcs) implements the shared objects that are manipulated by the application programs. It consists of a process at each node as well as possibly other hardware. Every object is assumed to have a *serial specification* (cf. [25]) defining a set of *(memory) operations*, which are ordered pairs of calls and responses, and a set of *(memory) operation sequences*, which are the allowable sequences of operations on that object. For example, in the case of a read/write object, the ordered pair [$\text{Read}_i(x)$, $\text{Return}_i(x, v)$] forms an operation (p_i reads v from x) for any p_i, x, and v, as does [$\text{Write}_i(x, v)$, $\text{Ack}_i(x)$] (p_i writes v to x). The set of operation sequences consists of all sequences in which every read operation returns the value of the latest preceding write operation (the usual read/write semantics). The interface to the mcs consists of *calls* (also called invocations) and *responses* on particular objects.

The *run-time environment* at a node takes as input the application program and executes instructions on the mcs. An *operation* is a specific instance of an execution of an instruction. A memory operation consists of two parts, a call (to the mcs process) and a matching response (from the mcs process). A control operation consists of an evaluation of its condition. A control operation is represented by the result (true or false) of the evaluation. Thus the run-time environment must keep track of the local state of the application process in order to perform the evaluation. The run-time environment and mcs process may also communicate concerning issues such as rollback and compensating operations, necessary to implement certain optimizations. This communication can be modeled with calls to and responses from the mcs process.

An *event* is a call, a response, or a control operation (condition evaluation). An *execution* (of the system) is a sequence of events such that there is a correspondence between calls and responses (matching object and process) and each response follows its corresponding call.

3.2. Sequential Executions. Although an execution is a sequence of events, it can also be viewed as containing operations. Each control operation is itself an event. The memory operations in an execution are obtained by matching up corresponding call and response events; we assume that the run-time environment matches the call and response events defining the memory operations. We also assume that the run-time environment identifies, possibly after the fact, a subset of all the operations executed. This subset consists of the operations that we want to consider as "really happening", and there must be an ordering of this subset that satisfies certain properties. Two important properties, which we present next, are satisfying the serial specifications of the objects (called "legal") and being consistent with a sequential execution of the program (called "admissible").

First, a piece of notation: if τ is a sequence of operations, and op_i precedes op_j in τ, we write

[2] Our framework does not restrict the data types; however, our programming techniques deal only with read/write operations.

A sequence τ of operations is *legal* if for each object x, $\tau|x$, the subsequence of τ consisting of exactly the operations involving x, is in the serial specification of x.

The notion of a sequential execution of the program is formalized with the notion of a flow control sequence for a process p_i. We build up inductively an execution of p_i's program in which every instruction finishes executing before the next one begins. Given process p_i's program, a *flow control sequence*, fcs_i, is a sequence of operations defined as follows. The first element of fcs_i is an instance of the first instruction in p_i's program. Suppose the k-th element of fcs_i, denoted *op*, is an instance of instruction I in the program. If *op* is a control operation and its condition evaluates to true, then the $(k+1)$-st element of fcs_i is an instance of the instruction whose label is the branch of instruction I. Otherwise the $(k+1)$-st element of fcs_i is an instance of the instruction immediately after I in the program. A flow control sequence can either be finite or infinite (for non-terminating programs). Note that the flow control sequence implies a total order on the operations appearing in it; we denote this order by $\overset{fcs_i}{\Longrightarrow}$.

Let fcs_i be a flow control sequence for p_i. A sequence τ of memory operations is *fully fcs_i-admissible* if $\tau|i$, the subsequence of τ consisting exactly of operations involving p_i, is equal to the subsequence of fcs_i consisting exactly of the memory operations. Intuitively, this implies that the ordering of operations by p_i in τ agrees with some flow control sequence fcs_i for p_i, and does not end unless the program terminates. A sequence τ of memory operations is *partially fcs_i-admissible* if $\tau|i$ is a prefix of the subsequence of memory operations in fcs_i. Intuitively, this implies that, so far, the ordering implied by the flow control sequence is obeyed by τ, but it is not necessarily completed yet.

A sequence τ of memory operations is *fully* (resp., *partially*) *admissible* with respect to a set of flow control sequences $\{fcs_i\}_{i=1}^{n}$, one for each p_i, if it is fully (resp., partially) fcs_i-admissible for all i.

A sequence of memory operations is a *sequential execution* if it is legal and fully admissible (with respect to some set of flow control sequences).

Claim 3.1. *Any legal partially admissible sequence of memory operations is a prefix of a sequential execution and vice versa.*

3.3. Weak and Strong Operations. In some consistency conditions it is possible to mark certain instructions in a program as *strong*; all other instructions are *weak*. An instance of a strong instruction is a *strong operation* and an instance of a weak instruction is a *weak operation*. (Strong and weak operations provide different levels of consistency and are part of the definition of hybrid consistency.) In the case of read/write objects this means that it is possible to use strong reads, strong writes, weak reads, and weak writes. In the rest of the paper, we use op_i to denote an operation invoked by p_i (weak or strong), and by sop_i we denote a strong operation invoked by process p_i. We use superscripts, e.g., $op_i^1, op_i^2, \ldots$, to distinguish between operations invoked by the same process (note that the superscript does *not* imply any ordering of the operations). We sometimes use a shorthand notation for read and write operations and denote by $r_i(x,v)$ a read operation (weak or strong) invoked by process p_i returning v from x; we denote by $w_i(x,v)$ a write operation (weak or strong) invoked by process p_i writing v to x. Similarly, $sr_i(x,v)$ is a strong read operation invoked by process p_i returning v from x; $sw_i(x,v)$ is a strong write operation invoked by process p_i writing v to x.

3.4. Control Operations and the Influence Relation. When defining and analyzing consistency conditions, it is important to take into account the effects of control operations. A specific example of the pitfalls associated with failing to do so appears in Section 4, when we present our new consistency conditions. To capture the effect of control operations, we define a partial order on operations in a flow control sequence; this partial order is featured in the consistency conditions presented below. Based on the relation fcs_i we define a partial order $\stackrel{co_i}{\Longrightarrow}$ called the *control order*. Formally, for any two memory operations op_i^1 and op_i^2, $op_i^1 \stackrel{co_i}{\Longrightarrow} op_i^2$ if there exists a control operation op_i^3 such that $op_i^1 \stackrel{fcs_i}{\Longrightarrow} op_i^3 \stackrel{fcs_i}{\Longrightarrow} op_i^2$.

We now formalize the notion of one operation influencing another, which relies on the control order. Let τ be a sequence of memory operations and let co_i be a partial order on the operations that is consistent with τ, for each p_i. An operation op_j^1 *directly influences* an operation op_k^2 *in* τ (with respect to the co_i's), if one of the following holds:

1. $op_j^1 \stackrel{co_j}{\Longrightarrow} op_k^2$ and op_j^1 is a read. (Note that $j = k$ in this case.) That is, op_j^1 is a read operation which could affect the execution of op_k^2 through a control operation.
2. $op_k^2 = r_k(y, v), op_j^1 = w_j(y, v), op_j^1 \xrightarrow{\tau} op_k^2$ and there does not exist $w_h(y, u)$ such that $u \neq v$ and $w_j(y, v) \xrightarrow{\tau} w_h(y, u) \xrightarrow{\tau} r_k(y, v)$. That is, op_k^2 is a read of the value written by op_j^1 and there is no intervening write of a different value.

The *influence* relation is the transitive closure of direct influence. Thus the influence relation is also defined with respect to a set of partial orders. Although we will not usually explicitly mention these partial orders, the influence relation will be used with admissible operation sequences and the relevant partial orders will be the control orders for the corresponding flow control sequences.

Note that an operation directly influences another operation only if it is ordered before it in τ. Since the influence relation is the transitive closure of direct influence, we have:

CLAIM 3.2. *If op_j^1 influences op_k^2 in τ, then $op_j^1 \xrightarrow{\tau} op_k^2$.*

The following lemma captures the intuition that if a read operation $op_j^1 = r_j(x, v)$ does not influence operation op_k^2, then op_k^2 would have been generated even if op_j^1 had read a value other than v.

LEMMA 3.3. *Let τ be a sequence of memory operations that is partially admissible with respect to a set of flow control sequences $\{fcs_i\}_{i=1}^n$. Let operation op_j^1 in τ be a read $r_j(x, v)$ that does not influence any operation in τ. Let τ' be the result of taking τ and changing op_j^1 to be $r_j(x, w)$ for some $w \neq v$. Then τ' is partially admissible for some set of flow control sequences $\{fcs_i'\}_{i=1}^n$.*

4. Defining Non-Sequential Consistency Conditions. In this section, we define three consistency conditions that generalize previously known ones for the non-sequential case. The reason they are generalizations is that in non-optimized systems, where operations at each process are invoked in program order and only one operation may be pending at a time, fcs_i is simply the sequence of operations in the order they were invoked. We then discuss (by an example) the importance of considering control operations (e.g., if-statements) when specifying consistency conditions.

Sequential consistency [28] is a strong consistency condition stating that there exists a sequential execution that is consistent with the way the actual execution appears to every process. Sequential consistency allows one to reason about a concurrent system using familiar uniprocessor techniques. Since uniprocessor systems are easier to reason about than concurrent systems, this is helpful. Thus many systems (try to) provide sequential consistency. However, providing sequential consistency can incur some costs. For instance, providing sequential consistency in message-based systems requires the response time of some operations to depend on the end-to-end message delay [14, 29]. Later in this paper, we show several programming techniques for achieving sequential consistency when the system only provides a weaker, and preferably cheaper, condition.

DEFINITION 4.1 (SEQUENTIAL CONSISTENCY). *An execution R is* sequentially consistent *if there exists a subset S of the memory operations in R, a set $\{fcs_i\}_{i=1}^{n}$ of flow control sequences, and a legal permutation τ of S such that τ is fully admissible with respect to $\{fcs_i\}_{i=1}^{n}$.*

Note that since τ is a permutation of S, and since τ is fully admissible with respect to $\{fcs_i\}_{i=1}^{n}$, S must include all operations that appear in all flow control sequences. Hence, S cannot just be chosen arbitrarily. In particular, unless the program is empty, S cannot be empty.

Weak consistency [12, 29] is a very relaxed condition; it does not impose any global ordering on any kind of operations. Its importance lies in the fact that it may be implemented very efficiently, and despite its weakness, there is a large class of programs for which it suffices. It requires that there exist a subset of the memory operations in the execution and a set of flow control sequences such that for each process, there is a legal permutation of those operations that is consistent with the process' own flow control sequence. Note that each processor may have a different permutation, or "view," of the operations.

DEFINITION 4.2 (WEAK CONSISTENCY). *An execution R is* weakly consistent *if there exist a subset S of the memory operations in R, and a set of flow control sequences $\{fcs_i\}_{i=1}^{n}$, one for each p_i, such that for each p_i, there exists a legal permutation τ_i of S that is fully fcs_i-admissible.*

Hybrid consistency [12] is intermediate between sequential and weak consistency; it combines the expressiveness of the former and the efficiency of the latter. Hybrid consistency distinguishes between two types of operations—strong and weak. It states that there must be a subset of the memory operations in the execution, a total order on the strong operations among them, and a set of flow control sequences satisfying the following. For each process, there is a legal permutation of the operations in the subset that is consistent with four orders: the process' own flow control sequence, every other process' control order, the total order on the strong operations, and the relative order of every pair of strong and weak operations by another process in that process' flow control sequence. Furthermore, all accesses of the same process to the same location will be viewed by all the processes in the same order. It is possible to implement hybrid consistency in such a way that weak operations are extremely fast [12].

DEFINITION 4.3 (HYBRID CONSISTENCY). *An execution R is* hybrid consistent *if there exist a subset S of the memory operations in R, a set of flow control sequences $\{fcs_i\}_{i=1}^n$, and a permutation ρ of the strong operations in S such that for each process p_i, there exists a legal permutation τ_i of S with the following properties:*

1. *τ_i is fully fcs_i-admissible; i.e., it is consistent with the process' own flow control sequence.*
2. *If $op_j^1 \overset{co_j}{\Longrightarrow} op_j^2$, then $op_j^1 \xrightarrow{\tau_i} op_j^2$, for any j;*[3] *i.e., it is consistent with every other process' control order,*
3. *If $op_j^1 \overset{fcs_j}{\Longrightarrow} op_j^2$ and at least one is strong, then $op_j^1 \xrightarrow{\tau_i} op_j^2$, for any j; i.e., it is consistent with the relative order of every pair of strong and weak operations by another process in that process' flow control sequence.*
4. *If $op_j^1 \xrightarrow{\rho} op_k^2$ (implying both are strong), then $op_j^1 \xrightarrow{\tau_i} op_k^2$, for any j and k; i.e., it is consistent with the total order on the strong operations.*
5. *If $op_j^1 \overset{fcs_j}{\Longrightarrow} op_j^2$ and op_j^1 and op_j^2 access the same location, then $op_j^1 \xrightarrow{\tau_i} op_j^2$, for any j; i.e., all accesses of the same process to the same location will be viewed by all the processes in the same order.*

We now discuss the last property in more detail. It states that all operations on the same object by the same process p_j are viewed by every other process in the same order as they are viewed by p_j. This property does not appear in the original definition of hybrid consistency [12]. However, it is necessary in order for some of our results to hold, as is shown in Section 6.3. Evidence suggests it is a reasonable assumption, since some previous authors make the even stronger assumption that all processes view all operations on the same object, no matter which process invoked them, in the same order.[4]

To illustrate the problem that occurs if hybrid consistency were defined without considering control operations, consider the following example, assuming x and y are initially 0 and all instructions are weak:

p_1's program	p_2's program
$\text{tmp}_1 := \text{read}(x)$;	$\text{tmp}_2 := \text{read}(y)$;
if $\text{tmp}_1 = 5$ then	if $\text{tmp}_2 = 5$ then
write$(y, 5)$;	write$(x, 5)$;

In this program, the values returned by the reads affect the decision whether to invoke the writes and the invocations of the writes affect the possible values returned by the reads. For example, assume this program is being executed on a hybrid consistent memory, and all operations are weak. If we ignore the existence of control instructions, the definition would allow the following execution R:

$$\begin{aligned} R|p_1 &= r_1^1(x,5), w_1^2(y,5) \\ R|p_2 &= r_2^1(y,5), w_2^2(x,5) \end{aligned}$$

[3] This condition is not part of the original definition of hybrid consistency [12], which did not include control operations.

[4] In [3, 22, 23], a total order on all the writes to the same location is assumed. In addition, it is assumed that a value read from a specific location can be uniquely identified with a write operation. Thus, all the processes view all the writes to the same location in the same order. Since a value read from a specific location can be uniquely identified with a write operation, each read is viewed by all the processes to be between the same writes to that location. These two assumptions imply that all the processes view all the operations on the same object in the same order.

$\underline{p_1\text{'s program}}$
tmp := read(x);
if tmp = v then
    write(y, u);
write(z, w);

FIG. 2. *Viewing order vs. execution order*

To show R is hybrid consistent we take

$$\begin{aligned} \tau_1 &= w_2^2(x,5), r_1^1(x,5), w_1^2(y,5), r_2^1(y,5) \\ \tau_2 &= w_1^2(y,5), r_2^1(y,5), w_2^2(x,5), r_1^1(x,5) \end{aligned}$$

Note that Condition 2 is not satisfied, since the sequences do not preserve the order of operations separated by control operations.

The program for p_1 allows the read(x) to return 5, whereas x could have taken the value 5 only if p_1 writes 5 to y, which will happen only if p_1 reads 5 in x. It is important to eliminate such executions with circular inference relations, where the prediction about the result of a control operation could affect its actual result.[5] If such behavior is allowed, then writing programs and arguing about them becomes almost impossible.

One might want to allow the programmer to choose whether a control operation should enforce an ordering or not. This can be done in a manner similar to the distinction we make between strong and weak memory operations. Introducing "weak" control operations in this manner further complicates the programming model; therefore, we have decided not to include them in our model.

4.1. Discussion of the Definitions.

Viewing Order vs. Execution Order:. We would like to emphasize that the order in which operations may be viewed by different processes need not reflect the order in which they are executed. For example, consider the program in Figure 2. Due to the control order, every process must view the write to z after the read from x. On the other hand, a sophisticated run-time environment (or compiler) could detect that the write to z would be invoked regardless of the outcome of the read from x. Therefore, the run-time environment can invoke the write to z before the read from x. This is allowed by the definition of hybrid consistency since it is possible to order the write to z after the read from x (even if they were executed in the reverse order).

Limitations of the Framework:. Our framework for defining consistency conditions does not support dependencies through registers and indirect addressing. For example, the program in Figure 3 is not handled by our framework. (We use & and * for indirect addressing in the style of C/C++.) Follow-up work [13] extends the framework to include RISC-type addressing, i.e., registers can be used to provide the data or the address for an operation (instead of just constants as in this paper). It shows that all the results of this paper hold in the extended framework as well, although the definitions are slightly more complicated.

[5] See [37] for more discussion of circular relations in the execution of programs.

p_1's program	p_2's program
$*\mathrm{tmp}_1 := \mathrm{read}(\&x)$;	$*\mathrm{tmp}_2 := \mathrm{read}(\&y)$;

FIG. 3. *Limitations of the framework*

p_1's program	p_2's program
swrite$(x, 1)$;	read(y);
swrite$(y, 1)$;	read(x);

FIG. 4. *The ordering of weak reads*

Programmability vs. Hardware Optimizations:. The definition of hybrid consistency includes several choices which reflect our opinions about the guarantees required by programmers from a consistency condition. These choices may disallow some hardware optimizations. We would like to stress, however, that these choices are not an inherent part of the framework. Specifically, if a particular choice we make does not allow an invaluable hardware optimization, then the definition of the consistency condition can be easily changed to support this optimization.

As an example, consider the program in Figure 4. In this program, the two writes by p_1 are labeled as strong while the two reads of p_2 are labeled as weak; assume that the initial values of x and y are 0. The definition of hybrid consistency requires that every process views its own weak reads in the same order as they appear in its flow control sequence. Hence, if the first read by p_2 returns 1 from y, then the second read by p_2 cannot return 0 from x.

We believe that this is the semantics expected by programmers. However, this implies that an implementation of hybrid consistency cannot afford to execute weak reads out-of-order unless it provides roll-back. This happens due to the fact that when the run-time system decides on the execution order for two weak reads by the same process, it may not know whether there are concurrent strong writes to the same objects (as in Figure 4). Yet, if it important to allow hardware optimizations that require a process to view its own weak reads out-of-order, then the definition of hybrid consistency can be easily changed by removing the first condition in Definition 4.3. We remark that in this case, the result about running data-race free programs (Theorem 6.5) remains correct, but the programming techniques developed in Section 5 are no longer valid.

5. Static Approach. In this section we present techniques for writing programs for hybrid consistent shared memories that are based on statically classifying accesses according to their type (read or write) and the object they access. In Section 5.1, we show that every hybrid consistent execution of a program in which all writes are strong is sequentially consistent. This result is used to develop efficient synchronization code in Section 5.2. In [11], we present the symmetric result, that every hybrid consistent execution of a program in which all reads are strong is sequentially consistent. In order to prove this theorem, further assumptions on the execution must be made. Although these assumptions are shown to be necessary, they make the result impractical.

5.1. Strong Writes. We prove the following theorem:

THEOREM 5.1. *Every hybrid consistent execution of a program in which all writes are strong and all reads are weak is sequentially consistent.*

Proof. Let R be such an execution. Let S be a subset of memory operations in R, fcs_i for each p_i be a flow control sequence, ρ be a permutation of the strong operations (namely, the writes) in S, and τ_i for each p_i be a legal permutation of S as guaranteed by the definition of hybrid consistency. We will insert the (weak) read operations into ρ to construct τ, a legal permutation of S that is fully admissible with respect to the fcs_i's.

A read by process p_i is inserted after any write that precedes it in τ_i and before any write that follows it in τ_i. This can be done since τ_i agrees with ρ on the order of strong operations. Furthermore, it will be inserted after any read by p_i that precedes it in τ_i and before any read by p_i that follows it in τ_i. The ordering of read operations by different processes is unimportant.

Clearly, τ is a permutation of S. Since τ_i preserves the ordering of operations by p_i, it follows that τ is fcs_i-admissible for all i.

The fact that τ is legal follows from the fact that τ_i is legal, that τ_i agrees with ρ on the order of strong operations and from the construction of τ. $\Box$

5.2. Using Strong Writes to Program with Critical Sections. Theorem 5.1 is useful for designing and proving correctness of programs which rely on hybrid consistency. A simple way to use it is to take a program which is designed for sequential consistency, label each write as a strong write and each read as a weak read, and run it on a hybrid consistent memory. However, there is even a more efficient way to employ the above theorem. If the program has explicit synchronization code dedicated to coordinating memory access operations, while the rest of the code ignores synchronization issues, then it is possible to apply Theorem 5.1 only to the synchronization code, and label all other memory accesses, both reads and writes, as weak. We demonstrate this method for mutual exclusion. Given a mutual exclusion algorithm designed for sequentially consistent memories, we produce a modified algorithm by adding one strong write (to a location which is never read from) to the entry section and by labeling all the writes in the synchronization part of the code as strong, while all other memory accesses are labeled as weak. We prove that the modified algorithm guarantees mutual exclusion (in a strong sense) on hybrid consistent memories.

Several papers on relaxed consistency conditions refer to the common method of programming with critical sections as a justification for the separation of strong and weak operations. In [21], it is argued (page 19):

> "For example, a large class of programs are written such that accesses to shared data are protected within critical sections. Such programs are called *synchronized programs*, whereby writes to shared locations are done in a mutually exclusive manner (no other reads or writes can occur simultaneously). In a synchronized program, all accesses (except accesses that are part of the synchronization constructs) can be labeled as $ordinary_L$."

Our result shows that there is a more efficient way to utilize synchronized programs. If the mutual exclusion algorithm is non-cooperative, then it is not necessary to label reads that are part of the synchronization constructs (i.e., entry and exit sections) as strong (provided that an extra write is added).

We remind the reader that an algorithm for mutual exclusion consists of four disjoint sections for each process—entry, critical, exit and remainder (cf. [36]). In the *entry* section, a process tries to gain access to the *critical* section; the *exit* section is executed by each process upon leaving the critical section; the *remainder* section is the rest of the code. Processes cycle through the four sections of their code. Informally, an algorithm guarantees mutual exclusion provided that:

Mutual exclusion: no two processes are inside the critical section at the same time, and

Deadlock freedom: if there exists a process that is continually in its entry section from some point on, then there exists another process that enters (and leaves) its critical section infinitely often.

A mutual exclusion algorithm provides code for the entry and exit sections. It should treat the code in the critical and remainder sections as black boxes (with some restrictions, as discussed below).

The above definition of mutual exclusion assumes that the order in which operations are executed reflects the order in which they are viewed by the processes. However, our formalism for defining consistency conditions puts restrictions only on the order in which operations are viewed, which may not correspond directly to the order in which they are executed. To be able to cope with these conditions, we define *logical mutual exclusion* as follows. Given a mutual exclusion algorithm (program), consider a flow control sequence for process p_i and let CS_i^k be the set of operations invoked by process p_i during the kth time that p_i executes the critical section in this sequence.

Algorithm $\mathcal{A}$ guarantees *logical mutual exclusion based on a consistency condition* C provided that the following holds. Let σ be an execution of $\mathcal{A}$ that is allowed by C and let $\{\tau_r\}_{r=1}^n$ be a set of sequences as required in the definition of C (the views of the different processes). Consider the CS_i^k's induced by $\stackrel{fcs_i}{\Longrightarrow}$.

Logical mutual exclusion: For any four operations $op_i^1, op_i^2 \in CS_i^k$ and $op_j^1, op_j^2 \in CS_j^l$, $op_i^1 \xrightarrow{\tau_q} op_j^1$ if and only if $op_i^2 \xrightarrow{\tau_s} op_j^2$, for all processes p_q and p_s. As before, this implies that there is a total order on all critical section executions; furthermore, this ordering is the same in all τ_r's.

Deadlock freedom: For every process p_q, if τ_q is infinite and there exists a process p_i that is in its entry section from some point on in τ_q, then there is another process p_j that enters (and leaves) its critical section infinitely often in τ_q.

In the rest of this section we mean logical mutual exclusion whenever we refer to mutual exclusion.

Our result assumes that the mutual exclusion algorithm does not communicate with the algorithm that is executed inside the critical section or remainder section. Specifically, we assume that variables that are accessed in the entry or exit section are not accessed in the critical or remainder sections. To capture this property, let the *exclusion set* of a mutual exclusion algorithm $\mathcal{A}$ be the set of shared variables read in the entry or exit sections of $\mathcal{A}$; this set is denoted $exc(\mathcal{A})$.

DEFINITION 5.1. *A mutual exclusion algorithm $\mathcal{A}$ is* non-cooperative *if every process which executes the critical section or the remainder section of $\mathcal{A}$ does not write any variable in $exc(\mathcal{A})$; otherwise, the algorithm is* cooperative.[6]

We assume that the mutual exclusion algorithm is designed to run with any code in the critical section (subject to the above restriction), and that it can be run in asynchronous systems (i.e., the algorithm does not depend on any timing behavior).

CLAIM 5.2. *There is at least one write to a variable in $exc(\mathcal{A})$ in the exit section of every non-cooperative mutual exclusion algorithm based on sequential consistency.*

Proof. Assume, by way of contradiction, that there exists a non-cooperative mutual exclusion algorithm $\mathcal{A}$ such that there is no write to any of the variables of $exc(\mathcal{A})$ in the exit section of $\mathcal{A}$. Assume that we run the algorithm with the following critical section (for every process p_i), under the assumption that x is initially 0:

$$tmp_i = r(x)$$
$$tmp_i := tmp_i + 1$$
$$w(x, tmp_i)$$

Recall that tmp_i and x are not accessed in the entry and exit sections of $\mathcal{A}$. Consider now a sequentially consistent execution R of $\mathcal{A}$ with this critical section in which p_0 starts at time 0 and runs until it enters the critical section, completes it, and exits; then p_1 starts and runs until it enters the critical section, completes it, and exits. Denote by t the time at which p_0 completes the read operation from x.

We claim that there is a sequentially consistent execution R' of $\mathcal{A}$ with this critical section in which p_0 starts at time 0 and runs until time t, when it completes the read operation; then p_1 starts and runs until it enters the critical section, completes it, and exits; then p_0 completes its critical section and exits. R' exists because we assumed the algorithm is asynchronous. Note that there is no write in the exit section of $\mathcal{A}$ and there is no write to variables in $exc(\mathcal{A})$ in the critical section of p_0. Thus, all the values read by p_1 in its entry section in R' are equal to the values it reads in its entry section in R; since in R, p_1 must eventually enter the critical section (to avoid deadlock) it must eventually enter the critical section in R'.

Clearly, R' violates the definition of mutual exclusion, since p_0 and p_1 are inside the critical section at the same time. Furthermore, note that in R' both p_0 and p_1 read the value 0 from x and thus the final value of x at the end of R' is 1. This final value could not be obtained in any execution that preserves logical mutual exclusion. □

Given a non-cooperative algorithm $\mathcal{A}$ for the mutual exclusion problem based on sequential consistency, label every write in the entry and exit sections as strong and every read in the entry and exit sections as weak; operations inside the critical section or remainder section are labeled as weak. Next, add a strong write to some object

[6] The definition of non-cooperative algorithms given in [12] allows processes that are executing the critical section to access variables in $exc(\mathcal{A})$. The additional requirement made here seems quite reasonable, since we usually want to treat the mutual exclusion algorithm as a general solution, independent of the rest of the code, and to use it as a subroutine. This is also the approach taken in many of the known solutions to the mutual exclusion problem [36]. A more detailed discussion and motivation for non-cooperative mutual exclusion algorithms appears in [12].

$nac(\mathcal{A})$ that is not accessed elsewhere in the program such that this write will be the last instruction executed (in a flow control sequence) before each critical section. Call this modified algorithm $\mathcal{A}'$.

In order to prove that $\mathcal{A}'$ guarantees mutual exclusion based on hybrid consistency, we have to show that it is free of deadlocks and that it guarantees logical mutual exclusion.

LEMMA 5.3. *$\mathcal{A}'$ guarantees deadlock freedom based on hybrid consistency.*

Proof. Assume, by way of contradiction, that there exists a hybrid consistent execution σ of $\mathcal{A}'$ that has a deadlock. Denote by $\{\tau_j\}_{j=1}^{n}$ the set of sequences of operations as guaranteed by the definition of hybrid consistency. Let σ' be the execution that results by eliminating from σ all operations that are not invoked inside (w.r.t. the flow control sequences) the entry or exit section of $\mathcal{A}'$. Since the algorithm is non-cooperative, there are no writes to variables in $exc(\mathcal{A}')$ outside (w.r.t. the flow control sequences) the entry and exit sections of $\mathcal{A}'$. Thus, σ' includes all the writes (in σ) to variables in $exc(\mathcal{A}')$, and thus, σ' is hybrid consistent. Specifically, define for each j a flow control sequence fcs'_j by eliminating from fcs_j all the operations of p_j that do not appear in σ'. Thus, the set of sequences $\{\tau'_j\}_{j=1}^{n}$ that results from eliminating all the operations that do not appear in σ' from $\{\tau_j\}_{j=1}^{n}$ obeys all the requirements in the definition of hybrid consistency (w.r.t. the $\{fcs'_j\}_{j=1}^{n}$). Note that all the writes in σ' are strong, and thus, by Theorem 5.1, σ' is sequentially consistent. Furthermore, in σ' there is a deadlock. Since there are no reads from $nac(\mathcal{A})$, the execution σ'' that results from eliminating all the writes to $nac(\mathcal{A})$ is also sequentially consistent and there is a deadlock in σ''. Note that σ'' is an execution of $\mathcal{A}$ in the case of an empty critical section. Thus, there is a sequentially consistent execution of $\mathcal{A}$ that has a deadlock. This is a contradiction. ☐

LEMMA 5.4. *$\mathcal{A}'$ guarantees logical mutual exclusion based on hybrid consistency.*

Proof. Recall that a sequential execution is a sequence of operations. Hence, each sequential execution of a program can be viewed by itself as a sequence τ of operations that obeys the requirements in the definition of sequential consistency. Therefore, since $\mathcal{A}$ guarantees mutual exclusion based on sequential consistency, every sequential execution of $\mathcal{A}$ must guarantee logical mutual exclusion as defined above. We will show that if $\mathcal{A}'$ does not guarantee mutual exclusion based on hybrid consistency, we may build a sequential execution of $\mathcal{A}$ in which mutual exclusion is violated.

Let σ be a hybrid consistent execution of an instance of $\mathcal{A}'$ in which every execution of the critical section consists of two writes, and let $\{\tau_j\}_{j=1}^{n}$ be the set of sequences of operations as guaranteed in the definition of hybrid consistency.

Call the last operation executed by a process before entering the critical section the *entry point.* Note that the entry point is a write operation on $nac(\mathcal{A})$. By Claim 5.2, there is at least one write in the exit section. Call the first write in the exit section the *exit point.* Each pair of matching entry and exit points is called a *critical pair.* We now show that in each τ_i all the critical pairs are ordered in a non-overlapping way, which gives the result for critical section executions.

Assume, by way of contradiction, that there exists a sequence τ_j in which two critical pairs overlap. Assume that $[op_k^1, op_k^2]$ and $[op_l^1, op_l^2]$ are critical pairs that overlap and assume, without loss of generality, that $op_k^2 \xrightarrow{\tau_j} op_l^2$. Let σ' be the execution that results by eliminating from σ all operations that are not invoked inside entry and exit sections of $\mathcal{A}'$. Since the algorithm is non-cooperative, σ' includes all the writes (in σ) to variables in $exc(\mathcal{A}')$, and thus, σ' is hybrid consistent. By Theorem 5.1, σ' is sequentially consistent. Thus, we may build a legal sequence of operations τ in the same way as in the proof of Theorem 5.1. That is, the strong writes are ordered in τ in the order they appear in any sequence τ_i. A read by process p_i is inserted after any write that precedes it in τ_i and before any write that follows it in τ_i. Furthermore, it will be inserted after any read by p_i that precedes it in τ_i and before any read by p_i that follows it in τ_i. The ordering of read operations by different processes in unimportant. Note that τ is a sequential execution of $\mathcal{A}'$ in the case of an empty critical section.

By the construction of τ, and since we assumed that $[op_k^1, op_k^2]$ and $[op_l^1, op_l^2]$ are overlapping in τ_j, the latest of op_k^1 and op_l^1 precedes both op_k^2 and op_l^2 in τ. Consider the prefix τ' of τ that ends with the latest of op_k^1 and op_l^1. Note that any operation in τ' that is invoked during the exit sections that correspond to $[op_k^1, op_k^2]$ and $[op_l^1, op_l^2]$ is a read. Remember that there are no reads from $nac(\mathcal{A})$. Thus, τ'', the result of eliminating from τ' all reads that are invoked during the exit sections that corresponds to $[op_k^1, op_k^2]$ and $[op_l^1, op_l^2]$ and all writes to $nac(\mathcal{A})$, is a prefix of a sequential execution of $\mathcal{A}$. Denote the writes in the critical section that corresponds to $[op_k^1, op_k^2]$ by w_k^1 and w_k^2 and the writes in the critical section that corresponds to $[op_l^1, op_l^2]$ by w_l^1 and w_l^2. Add w_k^1, w_l^1, w_k^2 and w_l^2 in this order to τ'', to form τ'''. Since all operations that were added to τ'' are writes, τ''' is legal and is therefore a prefix of a sequential execution. Moreover, $w_k^1 \xrightarrow{\tau'''} w_l^1$ but $w_l^1 \xrightarrow{\tau'''} w_k^2$, which is a violation of logical mutual exclusion. Thus, τ''' can be extended to a sequential execution of $\mathcal{A}$ that violates mutual exclusion. This is a contradiction to the assumption that every sequential execution of $\mathcal{A}$ guarantees logical mutual exclusion.

Thus, we have shown that all critical pairs appear in every sequence τ_j in a non-overlapping manner. Since the exit points are strong writes, all the processes agree (in the τ_j's) on the same non-overlapping order for the critical pairs. Hence, we only need to show that the order in which critical pairs are ordered in every sequence τ_j is unique.

Let cp, cp' and cp'' be three critical pairs such that cp is ordered before cp' and after cp'' in any τ_i. Since the exit point of cp is a strong operation, and since it is ordered before the entry point of cp', all the operations of cp are ordered before the entry point of cp'. Since the entry point of cp is a strong operation, and since it is ordered after the exit point of cp'', all the operations of cp are ordered after cp''. Thus, (logical) mutual exclusion is also guaranteed by the algorithm. □

Note that, in cooperative algorithms, a process that participates in the mutual exclusion protocol, but does not wish to enter the critical section, can access variables in the exclusion set while executing the remainder section. If these accesses are labeled as weak, the solution might not be correct anymore. A simple example (of a token passing mutual exclusion algorithm) is detailed in [11].

6. Running Data-Race-Free Programs. In this section we prove that data-race-free programs behave on hybrid consistent memory implementations as if they were sequentially consistent. Hybrid consistency is a weaker condition than sequential consistency, and can be implemented more efficiently [12, 14, 29]. Clearly, having data-race-free programs behave on hybrid consistent memory implementations as if they were sequentially consistent is a desirable property, since many concurrent programs attempt to be data-race-free. In our opinion, the proofs in this section are more transparent than proofs of similar results using other formalisms [22, 23]. This also demonstrates the usefulness of our framework for investigating and proving properties of consistency conditions in general (cf. [13]).

6.1. Definition of Data-Race-Free Programs. Let op_i^1 and op_j^2 be two operations appearing in some sequence of memory operations α. Then

- $op_i^1 \overset{po}{\Longrightarrow} op_j^2$ if $i = j$ and $op_i^1 \xrightarrow{\alpha} op_j^2$.
- $op_i^1 \overset{so}{\Longrightarrow} op_j^2$ if both op_i^1 and op_j^2 are strong operations and $op_i^1 \xrightarrow{\alpha} op_j^2$.

The relation *happens before*, denoted by $\overset{hb}{\Longrightarrow}$, is the transitive closure of the union of $\overset{po}{\Longrightarrow}$ and $\overset{so}{\Longrightarrow}$. This definition is similar to the definition of *happens before* in [3] and is closely related to the definition of *happened before* defined by Lamport [27] for message passing systems.

Two memory accesses *conflict* if they both access the same memory location and at least one of them is a write. A *data race* occurs in a sequence of memory operations when two conflicting memory accesses are not ordered by the happens before relation.

DEFINITION 6.1. *A program is* data-race-free *if none of its sequential executions contains a data race.*

These definitions are modeled after those in [3].

6.2. Running Data-Race-Free Programs on Hybrid Consistent Memory. We prove that every hybrid consistent execution of a data-race-free program is sequentially consistent.

To prove this result, we consider a legal sequence of memory operations τ_i, as guaranteed for some process p_i in the definition of hybrid consistency, that is *minimal* with respect to the number of *switched* operations (operations by the same process p_j whose order in τ_i is not consistent with p_j's flow control sequence). We show that if τ_i is not fully admissible (i.e., a sequential execution), then there exists a prefix of a sequential execution of the program that contains a data race. If τ_i is not fully admissible, it must contain at least one pair of switched operations. We locate the "first" pair of switched operations in τ_i, such that no other pair of switched operations is ordered between them. Because τ_i is minimal we know this pair was switched to preserve legality. This fact is used to show that there is a data race between some pair of operations that precedes this switched pair. Our main problem is to place these two operations (and the data race between them) in a legal and partially admissible sequence. This is done by taking the two operations and the operations that influence them and ordering them as in τ_i, and adding any operations necessary to preserve the flow control sequences of all processes. The key point to prove about the resulting sequence is its legality. In doing so, we either change the value that a read returns (and invoke Lemma 3.3), or, if this does not help, we show that there is a data race earlier in the sequence. Thus we have constructed a prefix of a sequential execution with a data race, which is a contradiction. The details follow.

Fix a data-race-free program `Prog`, a hybrid consistent execution R of `Prog`, and as guaranteed by the definition of hybrid consistency, a subset S of the memory operations and a flow control sequence fcs_i for each process p_i.

Fix an arbitrary process p_i. Let T be the set of all operation sequences τ_i that satisfy Definition 4.3 (hybrid consistency). For τ_i in T, an ordered pair of operations of p_j, $\langle op_j^2, op_j^1 \rangle$, is *switched* in τ_i if $op_j^2 \xrightarrow{\tau_i} op_j^1$, but $op_j^1 \overset{fcs_j}{\Longrightarrow} op_j^2$.

Let τ_i be some element of T with a minimal set of switches. That is, there does not exist $\tau_i' \in T$, such that the set of pairs of switched operations of all processes in τ_i' is strictly contained in the set of pairs of switched operations of all processes in τ_i.

We prove the main theorem of this section by way of contradiction, using the following lemma.

LEMMA 6.1. *If τ_i is not fully admissible, then there exists a prefix of a sequential execution of **Prog** which contains a data race.*

Proof. Assume that τ_i is not fully admissible. We first prove the following claim, which locates the pair of operations that is a candidate for a data race.

CLAIM 6.2. *There exist two operations $op_j^2(x,v)$ and $op_k^1(x,w)$ in τ_i such that*

1. $op_j^2(x,v) \xrightarrow{\tau_i} op_k^1(x,w)$,
2. *there is a data race between $op_j^2(x,v)$ and $op_k^1(x,w)$ in τ_i, and*
3. *there is no pair of switched operations in τ_i up to $op_k^1(x,w)$.*

Proof. Since τ_i is not fully admissible, there exists at least one pair of switched operations in τ_i. Let $\langle op_j^2(x,v), op_j^1(y,u) \rangle$ be the first ordered pair of switched operations, i.e., there is no other pair of switched operations which is completely ordered before $op_j^1(y,u)$ in τ_i. If there is more than one ordered pair that share the same second operation (namely, op_j^1), then choose the pair whose first operation is latest.

Since τ_i is minimal, $op_j^2(x,v)$ and $op_j^1(y,u)$ are switched in order to preserve legality.

Since the operations are by the same process p_j, and since the order of operations by the same process to the same object is preserved (last property in Definition 4.3), it follows that $x \neq y$. Thus, there exists an operation $op_k^1(x,w)$ which conflicts with $op_j^2(x,v)$ and $op_j^2(x,v) \xrightarrow{\tau_i} op_k^1(x,w) \xrightarrow{\tau_i} op_j^1(y,u)$.

If there is a strong operation by p_j between op_j^2 and op_j^1 (including one of them), then the third property in Definition 4.3 implies that they cannot be switched. Therefore, there are no strong operations by p_j between op_j^2 and op_j^1 in τ_i, including op_j^2 and op_j^1.

It is also the case that $j \neq k$. Suppose otherwise. Either op_j^2 and op_k^1 are switched or else op_k^1 and op_j^1 are switched. In the first case, $\langle op_j^2, op_k^1 \rangle$ would have been the pair chosen and in the second case, $\langle op_k^1, op_j^1 \rangle$ would have been the pair chosen.

Since $j \neq k$, op_j^2 and op_k^1 conflict, and there is no strong operation by p_j between them, there is a data race between op_j^2 and op_k^1 in τ_i. Thus the claim is shown. □

Let I be the set of operations in τ_i that influence either op_j^2 or op_k^1 (including op_j^2 and op_k^1 themselves). Let τ_i' be the shortest prefix of τ_i that includes every operation in I. Let π be the subsequence of τ_i' which contains exactly the set of all operations in I. The following claim shows that π is consistent with $\overset{fcs_l}{\Longrightarrow}$, for all l.

Claim 6.3. *For every pair of operations* op_l^1 *and* op_l^2, *if* $op_l^1 \xrightarrow{\pi} op_l^2$ *then* $op_l^1 \stackrel{fcs_l}{\Longrightarrow} op_l^2$.

Proof. By Claim 3.2, the influence relation is consistent with $\xrightarrow{\tau_i}$. Therefore, no operation in I follows op_k^1 in τ_i, implying that op_k^1 is the last operation in τ_i'. By Condition 3 in Claim 6.2, τ_i' does not include any pair of switched operations, and neither does π. □

Now, we construct a new partially admissible (with respect to the fcs_i's) sequence π' by adding to π every operation op_l^1 not in π, such that $op_l^1 \stackrel{fcs_l}{\Longrightarrow} op_l^2$ and op_l^2 is in π, for each process l, as follows. For every such operation that was originally in τ_i', add it in the same place as in τ_i'. For all other operations, add them arbitrarily, maintaining consistency with $\stackrel{fcs_l}{\Longrightarrow}$, for all l. By the definition of hybrid consistency, every added operation op_l^1 that is not in τ_i' must be weak. This is because op_l^1 precedes op_l^2 in $\stackrel{fcs_l}{\Longrightarrow}$, but follows op_l^2 in τ_i, a possibility precluded of any strong operation by hybrid consistency (Condition 3 in Definition 4.3).

We would now like to determine if π' is a legal sequence. By the definition of the influence relation and τ_i, π is a legal sequence. In particular, all reads in π are legal. Thus, if there are illegal reads in π', then each illegal read is either a read that was added to π, or a read that became illegal because of some write that was added to π.

Consider every illegal read r_l in π' that was added to π. We claim that r_l does not influence any operation in π', with respect to the co_i's. Suppose it does. Since r_l is a read, it must influence an operation through a control relation, i.e., $r_l \stackrel{co_l}{\Longrightarrow} op_l$, for some l. Since r_l is not in I, op_l is not in I. Thus op_l was included in π' because there exists op_l' in I such that $op_l \stackrel{fcs_l}{\Longrightarrow} op_l'$. This implies that $r_l \stackrel{co_l}{\Longrightarrow} op_l'$ and that r is in I. We therefore have a contradiction.

Now, for each of the illegal reads in π' that were added to π, change its value to match the value of the most recent write to the same object. (Note that this can be done since an illegal read in π' does not influence any other operation in π'.) Call this sequence π''. Apply Lemma 3.3 successively for each fixed-up read, starting with π', to deduce that π'' is partially admissible with respect to some set of flow control sequences $\{fcs_l'\}_{l=1}^n$. So, if there are no illegal reads in π'', then π'' is legal and it follows that it is a prefix of a sequential execution of `Prog`.

To complete the proof of the lemma when π'' is legal, we now show that π'' contains a data race between $op_j^2(x, v)$ and $op_k^1(x, w)$. By Claim 6.2, there is no strong operation by p_j between op_j^2 and op_k^1 inclusive in τ_i'; also, note that no strong operation was added to π. Also, $j \neq k$, thus op_j^2 and op_k^1 are not ordered by the happens-before relation.

We are left with the case in which there is an illegal read in π''. It became illegal due to the insertion of some write. Let $r_q^1(z, v_1)$ be the first illegal read in π'', and let $w_p^1(z, v_2)$, $v_1 \neq v_2$, be the corresponding inserted write. Denote by σ the shortest prefix of π'' which includes $r_q^1(z, v_1)$. Let σ' be the same sequence where $r_q^1(z, v_1)$ is replaced by $r_q^2(z, v_2)$. We complete the proof by showing:

Claim 6.4. σ' *is a prefix of a sequential execution of* `Prog` *which contains a data race between* r_q^2 *and* w_p^1.

Proof. We first show that $r_q^1 \xrightarrow{\tau_i} w_p^1$. Assume otherwise that $w_p^1 \xrightarrow{\tau_i} r_q^1$. Since w_p^1 is not in I, it must not influence r_q^1 in τ_i. Therefore, there is another write $w_r^2(z, v_1)$ ordered between w_p^1 and r_q^1 which influences r_q^1 in τ_i. But, then w_p^1, w_r^2 and r_q^1 would be ordered similarly in π'. This gives a contradiction to the assumption that r_q^1 becomes illegal due to the insertion of w_p^1.

So we can assume that $r_q^1 \xrightarrow{\tau_i} w_p^1$; it follows that w_p^1 is not in τ_i', implying that w_p^1 is weak. We claim that $p \neq q$. Otherwise, if $p = q$, then $w_p^1(z, v_2) \overset{fcs_p}{\Longrightarrow} r_q^1(z, v_1)$. By the last property of hybrid consistency, it follows that $w_p^1(z, v_2) \xrightarrow{\tau_i} r_q^1(z, v_1)$, a contradiction. Therefore, $p \neq q$.

We now show that $r_q^1(z, v_1)$ and $w_p^1(z, v_2)$ are not ordered by the happens before relation in π'', and hence, they are not ordered by the happens before relation in σ. Suppose they were. Then, there is a strong operation sop_p in between w_p^1 and r_q^1 in π''. Therefore, $w_p^1 \overset{fcs_p}{\Longrightarrow} sop_p$ and thus $w_p^1 \xrightarrow{\tau_i} sop_p$. Therefore, sop_p is not in τ_i' since w_p^1 is not in τ_i', and hence is weak, a contradiction. Therefore, $r_q^1(z, v_1)$ and $w_p^1(z, v_2)$ are not ordered by the happens before relation in π'', and hence, in σ, and there exists a data race between them in σ. It follows that σ' contains a data race.

Now, σ' is a legal sequence of operations due to the change in the value returned by the read, implied by replacing r_q^1 with r_q^2. Since $r_q^2(z, v_2)$ is the last operation in σ', it clearly does not influence any operation in σ'. By Lemma 3.3, σ' is partially admissible for some set of flow control sequences $\{fcs_l''\}_{l=1}^n$. Therefore, σ' is a prefix of sequential execution of `Prog` which contains a data race between $r_q^2(z, v_2)$ and $w_p^1(z, v_2)$, as needed. □

Hence, we have shown that if τ_i is not fully admissible, then there exists a prefix of a sequential execution of `Prog` which contains a data race. □

The proof of the following theorem is now immediate.

THEOREM 6.5. *Every hybrid consistent execution of a data-race-free program is sequentially consistent.*

Proof. Let R be a hybrid consistent execution of a data-race-free program `Prog`. Fix a subset S of the memory operations in R and a set of flow control sequences fcs_i from the definition of hybrid consistency. Choose a process p_i. Let τ_i be a minimal legal permutation of S as guaranteed by the definition of hybrid consistency. Since `Prog` is data-race-free, no sequential execution has a data race. Then by Lemma 6.1, τ_i is fully admissible. Thus, R is a sequentially consistent execution of `Prog`. □

6.3. Reordering Process' Operations. The original definition of hybrid consistency in [12] did not include the last property, i.e., that all writes by the same process to the same object appear in the views of all other processes in the order implied by the flow control sequence of their invoking process. (The original definition also did not consider control operations.) Thus two weak operations by the same process p_i accessing the same location could be viewed by other processes in a different order from the order in which they appear in fcs_i. We show that under this behavior it is not true that every hybrid consistent execution of a data-race-free program is sequentially consistent.

Consider the following data-race-free program, assuming x and y are initially 0 and all instructions are weak:

p_1's program	p_2's program
$\mathrm{tmp}_1 := \mathrm{read}(x)$;	$\mathrm{tmp}_2 := \mathrm{read}(y)$;
$\mathrm{write}(x, 5)$;	$\mathrm{write}(y, 5)$;
if $\mathrm{tmp}_1 = 5$ then	if $\mathrm{tmp}_2 = 5$ then
$\quad\mathrm{write}(y, 5)$;	$\quad\mathrm{write}(x, 5)$;

Consider the following execution R.

$$\begin{aligned} R|p_1 &= r_1^1(x,5), w_1^2(x,5), w_1^3(y,5) \\ R|p_2 &= r_2^1(y,5), w_2^2(y,5), w_2^3(x,5) \end{aligned}$$

Although R is not sequentially consistent (we leave it to the reader to verify this), it is hybrid consistent:

$$\begin{aligned} \tau_1 &= w_2^2(y,5), r_2^1(y,5), w_2^3(x,5), r_1^1(x,5), w_1^2(x,5), w_1^3(y,5) \\ \tau_2 &= w_1^2(x,5), r_1^1(x,5), w_1^3(y,5), r_2^1(y,5), w_2^2(y,5), w_2^3(x,5) \end{aligned}$$

7. Discussion. As the demand for powerful computers grows faster than the technology to develop new processors, the need for highly parallel multiprocessors increases. However, in order to fully utilize such machines, convenient paradigms for writing concurrent programs must be developed. These paradigms should allow the user to enjoy the same simple model of the world as in uniprocessors, without sacrificing the performance of the whole system. These two goals are somewhat contradictory. Recent results indicate that there is a tradeoff between the similarity of a distributed shared memory to real shared memory, and the efficiency of the hardware.

In this paper we have tried to bridge over these two contradictory goals. We presented a general framework which encompasses the functionality of the compiler and the run-time system and models their interaction with the memory consistency system. Our framework allows the definition of known consistency conditions to be combined with implementations that exploit optimizations for reducing the latency of memory accesses. To the best of our knowledge, our definitions are unique in explicitly modeling the whole program, rather than just looking at the memory operations in isolation.

We also characterized requirements on programs that guarantee that they will behave on hybrid consistent memories as if they are sequentially consistent. This allows programmers to reason about certain classes of programs assuming sequential consistency, yet run them on more efficient hardware. The approaches we studied were (1) labeling all writes as strong and all reads as weak, (2) using the previous scheme to implement efficient mutual exclusion, and (3) running data-race-free programs. Note that the first two approaches do not necessarily yield data-race-free programs.

In this paper, we have assumed that the application program at each node is a serial code or, in other words, consists of a single thread of execution. We believe our framework and results can be naturally extended to support multi-threading as well. We briefly outline one approach. Instead of a flow control sequence, define a *flow control (partial) order* based on two new control operations, *fork* and *join*. A fork operation splits the code into several parallel threads of execution and a join operation joins several parallel threads of execution into one thread of execution. Operations that belong to the same thread of execution are ordered as in a flow control sequence and after the previous fork operation (if there exists one). Operations that belong to different parallel threads are not ordered by the flow control order. All operations that belong to threads that were joined by a join operation are ordered before the join operation that joined their threads. The definitions of the consistency conditions remain the same, with the one exception that we use the flow control orders instead of the flow control sequences.

This work is part of an on-going attempt to understand consistency conditions and their implications on programming, compiler design and architecture. Much research is still needed before this goal can be met. While more efficient, fault-tolerant algorithms for implementing various consistency conditions still need to be developed, our paper takes a complementary approach: it provides a clean and formal framework for investigating systematic methods, rules and compiler techniques to transform programs written for strong consistency conditions into correct programs for weaker consistency conditions.

Acknowledgments:. We would like to thank Kourosh Gharachorloo, Phil Gibbons, Martha Kosa and Michael Merritt for helpful comments. The anonymous referees made helpful comments which improved the presentation.

REFERENCES

[1] R. Acosta, J. Kjelstrup, and H. Torng. An Instruction Issuing Approach to Enhancing Performance in Multiple Functional Unit Processors. *IEEE Transanctions on Computers*, C-35(9):815–828, September 1986.

[2] S. Adve. Designing Memory Consistency Models for Shared-Memory Multiprocessors. Technical Report 1198, Computer Science Department, University of Wisconsin-Madison, December 1993. Ph.D. Thesis.

[3] S. Adve and M. Hill. Weak Ordering—A New Definition. In *Proc. of the 17th International Symposium on Computer Architecture*, pages 2–14, May 1990.

[4] S. Adve and M. Hill. A Unified Formalization of Four Shared-Memory Models. Technical Report 1051, Computer Science Department, University of Wisconsin, Wisconsin-Madison, September 1991.

[5] S. Adve and M. Hill. Sufficient Conditions for Implementing the Data-Race-Free-1 Memory Model. Technical Report 1107, Computer Science Department, University of Wisconsin, Wisconsin-Madison, September 1992.

[6] S. Adve, M. Hill, B. Miller, and R. Netzer. Detecting Data Races on Weak Memory Systems. In *Proc. of the 18th International Symposium on Computer Architecture*, pages 234–243, May 1991.

[7] Y. Afek, G. Brown, and M. Merritt. A Lazy Cache Algorithm. *ACM Trans. on Programming Languages and Systems*, 15(1):182–205, January 1993.

[8] M. Ahamad, R. Bazzi, R. John, P. Kohli, and G. Neiger. The Power of Processor Consistency. In *Proc. of the 5th ACM Symposium On Parallel Algorithms and Architectures*, pages 251–260, June/July 1993.

[9] M. Ahamad, J. Burns, P. Hutto, and G. Neiger. Causal Memory. In *5th International Workshop on Distributed Algorithms*, Greece, October 1991.

[10] M. Ahamad, G. Neiger, P. Kohli, J. Burns, and P. Hutto. Causal Memory: Definitions, Implementation, and Programming. *Distributed Computing*, 9(1), 1993.

[11] H. Attiya, S. Chaudhuri, R. Friedman, and J. Welch. Shared Memory Consistency Conditions for Non-Sequential Execution: Definitions and Programming Strategies. Technical Report LPCR #9306, Department of Computer Science, The Technion, June 1993.

[12] H. Attiya and R. Friedman. A Correctness Condition for High-Performance Multiprocessors. *SIAM Journal of Computing*, to appear. Also: Technical Report #767, Department of Computer Science, The Technion.

[13] H. Attiya and R. Friedman. Programming DEC-Alpha Based Multiprocessors the Easy Way. In *Proc. 6th ACM Symp. on Parallel Algorithms and Architectures*, pages 157–166, June 1994. Also: Technical Report LPCR #9411, Department of Computer Science, The Technion.

[14] H. Attiya and J. Welch. Sequential Consistency versus Linearizability. *ACM Transactions on Computer Systems*, 12(2):91–122, May 1994.

[15] R. Bisiani, A. Nowatzyk, and M. Ravishankar. Coherent Shared Memory on a Distributed Memory Machine. In *Proc. International Conf. on Parallel Processing*, pages I–133–141, 1989.

[16] J.-D. Choi and S. L. Min. Race Frontier: Reproducing Data Races in Parallel Program Debugging. In *Proc. of the 3rd ACM Symposium on Principles and Practice of Parallel Programming*, pages 145–154, April 1991.

[17] A. Dinning and E. Schonberg. Detecting Access Anomalies in Programs with Critical Sections. In *Proc. of the ACM Workshop on Parallel and Distributed Debugging*, pages 85–96, May 1991.

[18] M. Dubois and C. Scheurich. Memory Access Dependencies in Shared-Memory Multiprocessors. *IEEE Trans. on Software Engineering*, 16(6):660–673, June 1990.

[19] M. Dubois, C. Scheurich, and F. A. Briggs. Synchronization, Coherence and Event Ordering in Multiprocessors. *IEEE Computer*, 21(2):9–21, February 1988.

[20] J. Fisher. Very Long Instruction Word Architectures and the ELI-512. In *Proc. of the 10th International Symposium on Computer Architecture*, pages 140–150, 1991.

[21] K. Gharachorloo, D. Lenoski, J. Laudon, P. Gibbons, A. Gupta, and J. Hennessy. Memory Consistency and Event Ordering in Scalable Shared-Memory Multiprocessors. In *Proc. of the 17th International Symposium on Computer Architecture*, pages 15–26, May 1990.

[22] P. Gibbons and M. Merritt. Specifying Non-Blocking Shared Memories. In *Proc. of the 4th ACM Symp. on Parallel Algorithms and Architectures*, pages 306–315, July 1992.

[23] P. Gibbons, M. Merritt, and K. Gharachorloo. Proving Sequential Consistency of High-Performance Shared Memories. In *Proc. of the 3rd ACM Symposium on Parallel Algorithms and Architectures*, pages 292–303, July 1991.

[24] J. Hennessy and D. Patterson. *Computer Architecture: A Quantitative Approach*, pages 251–349. Morgan Kaufmann Publishers Inc., 1990.

[25] M. Herlihy and J. Wing. Linearizability: A Correctness Condition for Concurrent Objects. *ACM Trans. on Programming Languages and Systems*, 12(3):463–492, 1990.

[26] M. S. Lam and R. P. Wilson. Limits of Control Flow on Parallelism. In *Proc. of the 19th International Symposium on Computer Architecture*, pages 46–57, May 1992.

[27] L. Lamport. Time, Clocks and the Ordering of Event in a Distributed System. *Communications of the ACM*, 21(7):558–565, 1978.

[28] L. Lamport. How to Make a Multiprocessor Computer that Correctly Executes Multiprocess Programs. *IEEE Trans. on Computers*, C-28(9):690–691, 1979.

[29] R. Lipton and J. Sandberg. PRAM: A Scalable Shared Memory. Technical Report CS-TR-180-88, Computer Science Department, Princeton University, September 1988.

[30] J. Mellor-Crummey. On-the-fly Detection of Data Races for Programs with Nested Fork-Join Parallelism. *Supercomputing*, pages 24–33, November 1991.

[31] R. Netzer. Race Condition Detection for Debugging Shared-Memory Parallel Programs. Technical Report 1039, Computer Science Department, University of Wisconsin-Madison, August 1991. Ph.D. Thesis.

[32] R. Netzer and B. Miller. Improving the Accuracy of Data Race Detection. In *Proc. of the 3rd ACM Symposium on Principles and Practice of Parallel Programming*, pages 133–144, April 1991.

[33] R. Netzer and B. Miller. What are Race Conditions? Some Issues and Formalizations. *ACM Letters on Programming Languages and Systems*, 1(1), March 1992.

[34] Y. Patt, W. Hwu, and M. Shebanow. HPS, a New Microarchitecture: Rationale and Introduction. In *Proc. of the 18th Annual Microprogramming Workshop*, pages 103–108, December 1985.

[35] A. Peleg and U. Weiser. Future Trends in Microprocessors: Out-of-Order Execution, Speculative Branching and their CISC Performance Models. In *Proc. of the 17th Convention of Electrical and Electronics Engineers in Israel*, pages 263–266, May 1991.

[36] M. Raynal. *Algorithms for Mutual Exclusion*. MIT Press, 1986.

[37] D. Shasha and M. Snir. Correct and Efficient Execution of Parallel Programs that Share Memory. *ACM Trans. on Programming Languages and Systems*, 10(2):282–312, April 1988.

[38] A. Singh. A Framework for Programming using Non-atomic Variables. Technical Report TRCS-93-11, Department of Computer Science, University of California at Santa Barbara, July

1993.
[39] B. Smith. A Massively Parallel Shared Memory Computer. In *3rd ACM Symposium on Parallel Algorithms and Architectures*, July 1991. invited lecture.
[40] J. Smith. Dynamic Instruction Scheduling and the Astronautics ZS-1. *IEEE Computer*, pages 21–35, July 1989.
[41] R. N. Zucker and J.-L. Baer. A Performance Study of Memory Consistency Models. In *Proc. of the 19th International Symposium on Computer Architecture*, pages 2–12, May 1992.