Shared Memory Consistency Conditions for Non-Sequential Execution: Definitions and Programming Strategies

(EXTENDED ABSTRACT)

Hagit Attiya* Soma Chaudhuri† Roy Friedman* 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 however can 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 four known consistency conditions—sequential consistency, hybrid consistency, weak consistency, and release consistency—for non-sequential execution.

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 and second techniques statically classify accesses based on their type. This approach is extremely simple to use and leads to a general methodology for writing efficient synchronization code. The third technique is to avoid data races in the program; this technique also works on a simple variant of release consistent hardware, with an appropriate change to the definition of data race.

*Department of Computer Science, The Technion, Haifa 32000, Israel. Email: hagit@cs.technion.ac.il and roy@cs.technion.ac.il. Partially supported by B. and G. Greenberg Research Fund (Ottawa), by Technion V.P.R. funds, and by the fund for the promotion of research at the Technion.

†Laboratory for Computer Science, Massachusetts Institute of Technology, Cambridge, MA 02139. Email: soma@theory.lcs.mit.edu. On leave from the Department of Computer Science, Iowa State University. Supported in part by NSF grant CCR-89-15206, DARPA contracts N00014-89-J-1988 and N00014-92-J-4033, ONR contract N00014-91-J-1046, and a grant from the ISU Graduate College.

‡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. Supported in part by NSF grant CCR-9010730, an IBM Faculty Development Award, an NSF Presidential Young Investigator Award CCR-9158478, and TAMU Engineering Excellence funds.

ACM-SPAA'93-6/93/Velen,Germany.

1 Introduction

Overview: Intense interest is currently being focused on shared memory multiprocessors in many disciplines of computer science. 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 inter-processor communication delay, which increases with the number of processors, and the time to execute memory operations locally.

Many computer architecture techniques have been developed 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[1]. Specific instances include, e.g., [1, 17, 24, 34, 35, 36, 37, 38]. Since all these techniques deviate from the sequential order of memory accesses specified by the program, we call them "non-sequential".

Non-sequential execution of memory accesses complicates the user's model of memory. A consistency condition for shared memory specifies what guarantees are provided about the values returned in the presence of

[1] Predict the outcome of future conditional branches and begin memory accesses for the predicted branch.

concurrent accesses. A variety of consistency conditions have previously been proposed for shared memory architectures, e.g., [8, 7, 2, 3, 10, 16, 22, 20, 25, 27, 28]. In the presence of non-sequential execution, an obvious question is whether they can still be provided efficiently, or, if not, what guarantees are provided by the optimized system. A further question is how to program these optimized systems with new consistency conditions in a safe and effective manner.

In this paper, we present a theoretical foundation that allows programming to exploit non-sequential execution of memory accesses on multiprocessors. Such a foundation provides a common ground and interface between researchers investigating multiprocessor architectures, concurrent programming languages and parallelizing compilers. We extend four known consistency conditions to allow for non-sequential execution of memory accesses. Our conditions are stated as properties of executions, not hardware implementations. We then present and prove correct several techniques for programming on optimized architectures so as to provide the illusion of sequential consistency, which is easier to reason about. For hybrid consistency, the first approach is to label all writes (or all reads) as strong. The second method, again for hybrid consistency, is to avoid data races. An analogous result is shown for asymmetric hybrid consistency, our attempt at a formalization of release consistency.

Detailed Description: Our first contribution is a framework for defining consistency conditions that is general enough to allow non-sequential execution of memory operations. (See Sec. 2.) This framework is novel in combining the following two features. (1) The interface at which conditions are specified in our model is between the program and the system. This is the interface used in [27, 25, 10] and the natural one to use for specification to be independent of implementation. (2) The framework allows for arbitrary optimizations by the system, including especially non-sequential execution of memory accesses. The framework is then used to extend four 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 system*. (See Fig. 1.) An application program contains instructions to access shared memory

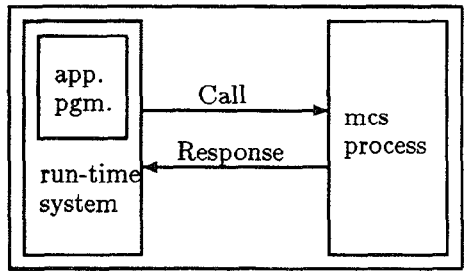

Figure 1: A node

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 system executes the shared memory instructions by interacting with the local mcs process; it bases its decisions as to which instructions to execute on the application program at that node. (We abuse the term run-time system and use it 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.)

A straightforward run-time system 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 system might submit operations out of order, might have multiple operations pending at a time, and might anticipate branches (sometimes incorrectly). We are not interested in the specific algorithm used by the run-time system. (That is another very interesting problem, beyond the scope of this paper.) Instead, our goal is to model the run-time system sufficiently abstractly so that any of a large number of specific run-time systems can fit into this framework. Obviously the run-time system cannot do just anything—the optimizations that it performs should be transparent to the application program. The condition we require of a correct run-time system is that there exists a way (after the fact) to take at least some of the operations performed by the run-time system and order them to be consistent with some "sequential execution" of the program. (Some of the operations performed by the run-time system 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 system to communicate with the mcs in order to perform other operations on the data, not requested by the application programs, but necessary for restoring the state of the shared variables due to incorrect predictions, for example. These operations are ignored when the subset of operations consistent with a sequential execution is taken.

Given this framework, we generalize four known consistency conditions—sequential consistency, weak consistency, hybrid consistency, and release consistency. (See Sec. 3.)

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 [10]; it unifies and generalizes several other consistency conditions appearing in the literature [2, 12, 16]. Memory access operations are classified as either *strong* or *weak*. A global ordering is imposed on strong operations at different processes, but not much is guaranteed about the ordering of weak operations at different processes, except for what is implied by their interleaving with the strong operations.

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. 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 present two different approaches for turning programs written for sequential consistency into programs that work for hybrid consistency. We believe that these techniques are a first step in developing automatic optimization techniques for the compilation and execution of parallel programs.

The first approach we present is based on statically labeling specific accesses as strong, according to their type. (See Sec. 4.) We derive two specific techniques from this approach. First, we prove that programs in which all writes are strong run on hybrid consistent shared memory implementations as if they were sequentially consistent. We then show how this result can be used to produce efficient synchronization code based on mutual exclusion. Our second static technique for programming with hybrid consistency is symmetric to the first: We show that, under certain assumptions, programs in which all reads are strong run on hybrid consistent shared memory implementations as if they were sequentially consistent.

The second general approach for programming with hybrid consistency is to run data-race-free programs. (See Sec. 5.) (This is analogous to the work of [2, 3, 22, 21].) 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., [5, 13, 14, 15, 29, 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.

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 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 operation. These methods are especially well-suited to applications in which reads greatly outnumber writes (or vice versa).

A variant of hybrid consistency that distinguishes between *release* and *acquire* operations has been proposed (e.g., [3, 21, 22, 20]). Informally, release and acquire operations are strong operations, but their effect is not symmetric—a release operation orders all weak operations by the same process that precede it, while an acquire operation orders all weak operations by the same process that follow it. Although recent research [19, 39] suggests that the benefits from this further classification are somewhat limited (when compared to hybrid consistency), in order to demonstrate the flexibility of our framework, and for compatibility with recent trends in the architecture community, we show in Sec. 6 that "asymmetric" data-race-free programs run on asymmetric hybrid consistent memory implementations as if they were sequentially consistent.

Sec. 7 compares our results with related work. We conclude with a discussion of the results in Sec. 8. Due to space limitations, many details are omitted; they may be found in [9].

2 Framework

System Components: Each *application program* consists of a sequence of instructions, each with a

unique label. There are two types of instructions, (shared) memory instructions and control instructions. A *memory instruction* specifies an access to a shared object. 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. The interface to the mcs consists of *calls* (or invocations) and *responses* on particular objects.

The *run-time system* takes as input the application program and executes instructions on the mcs[2]. It consists of a process at each node. An *operation* is a specific instance of an execution of an instruction. A memory operation consists of two parts, a call (to the mcs) and a matching response (from the mcs). 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 system must keep track of the local state of the application process in order to do the evaluation.

An *event* is a call, a response, or a control operation (condition evaluation). A *run* (of the run-time 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.

Sequential Executions: The memory operations in a run are obtained by matching up corresponding call and response events; each control operation is itself an event. We are interested in picking out a subset of the operations in a run, finding an ordering of those operations, and seeing what properties that ordering satisfies. To that end, we next define several properties on sequences of operations. The two main properties are satisfying the serial specifications of the objects (called "legal") and being consistent with a sequential execution of the program (called "admissible").

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.

[2]The run-time system and mcs may also communicate concerning issues such as rollback and compensating operations, necessary to implement certain optimizations. This communication can also be modeled with calls and responses.

The notion of a sequential execution of the program is formalized with the notion of a flow control sequence for a process p_i. Given process p_i's program, a *flow control sequence*, fcs_i, is a sequence of operations defined inductively as an execution of p_i's program in which every instruction finishes executing before the next one begins. (See the full paper for a more formal definition.) We denote by $\stackrel{fcs_i}{\Longrightarrow}$ the total order imposed by fcs_i. A sequence τ of memory operations is *fully fcs_i-admissible* if the ordering of operations by p_i in τ agrees with fcs_i and does not end unless the program terminates. A sequence τ of memory operations is *partially fcs_i-admissible* if 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 2.1 *Any legal partially admissible sequence of memory operations is a prefix of a sequential execution and vice versa.*

Weak and Strong Operations: It is possible to mark some 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*. In the sequel, op_i denotes an operation, weak or strong, invoked by p_i. Superscripts, e.g., $op_i^1, op_i^2, \ldots$, distinguish between operations invoked by the same process but do *not* imply any ordering of the operations. More explicitly, $r_i(x,v)$ denotes a read operation invoked by process p_i returning value v from variable x, and $w_i(x,v)$ denotes a write operation invoked by process p_i writing v to x.

Control Operations and the Influence Relation: To capture the effect of control operations, we define a partial order $\stackrel{co}{\Longrightarrow}$, called the *control order*, on operations in a flow control sequence fcs_i: $op_i^1 \stackrel{co}{\Longrightarrow} op_i^2$ 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 for each p_i, let co_i be a partial order on the operations that is consistent with τ. An operation op_l^1 *directly influences* an operation op_m^2 *in* τ (with respect to the co_i's), if one of the following holds:

1. $op_l^1 \overset{co_l}{\Longrightarrow} op_m^2$ and op_l^1 is a read. (Note that $l = m$ in this case.) That is, op_l^1 is a read operation which could affect the execution of op_l^2 through a control operation.

2. $op_m^2 = r_m(y, v), op_l^1 = w_l(y, v)$, $op_l^1 \xrightarrow{\tau} op_m^2$ and there does not exist $w_p(y, u)$ such that $u \neq v$ and $w_l(y, v) \xrightarrow{\tau} w_p(y, u) \xrightarrow{\tau} r_m(y, v)$.[3] That is, op_m^2 is a read of the value written by op_l^1 and there is no intervening write of a different value.

The *influence* relation is the transitive closure of direct influence.

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

Lemma 2.2 *Let τ be a sequence of memory operations that is partially admissible with respect to a set of flow control sequences $\{fcs_j\}_{j=1}^n$. Let operation op_l^1 in τ be a read $r_l(x, v)$ that does not influence any operation in τ. Let τ' be the result of taking τ and changing op_l^1 to be $r_l(x, w)$ for some $w \neq v$. Then τ' is partially admissible for some set of flow control sequences $\{fcs'_j\}_{j=1}^n$.*

3 The Consistency Conditions

In this section, we define three consistency conditions that generalize previously known ones for the non-sequential case; a fourth condition is presented in Sec. 6. 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.

Sequential consistency [27] is a strong consistency condition stating that there exists a sequential execution that is consistent with the way the actual run appears to every process. Providing sequential consistency in message-based systems requires the response time of some operations to depend on the end-to-end message delay [28, 11].

Definition 3.1 (Sequential consistency) *A run 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 $\{fcs_i\}_{i=1}^n$.*

[3] $op_i \xrightarrow{\tau} op_j$ means op_i precedes op_j in the sequence τ.

Weak consistency [28, 10] does not impose any global ordering on operations. It may be implemented very efficiently, and despite its weakness, there is a large class of programs for which it is sufficiently expressive. It requires that there exist a subset of the memory operations in the run and a set of flow control sequences such that for each process, there is a legal permutation of those operations that is consistent with both the process' flow control sequence and every other process' control order.

Definition 3.2 (Weak consistency) *A run R is* weakly consistent *if there exists a subset S of the memory operations in R, 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 with the following properties.*

1. *τ_i is fully fcs_i-admissible.*
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.*

Hybrid consistency [10] 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 run, 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' flow control sequence, every 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 [10].

Definition 3.3 (Hybrid consistency) *A run R is* hybrid consistent *if there exists 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.*
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. *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.*

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*.*

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*.*

The full paper includes an illustration of the problems (namely, circular dependencies) that can occur if the second property is not included. The fifth property 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 [10]. However, it is necessary in order for some of our results to hold, as is shown in the full paper. 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]

4 Static Approach

In this section we discuss techniques for writing programs for hybrid consistent shared memories that are based on statically classifying accesses depending on whether they are reads or writes.

In the full paper we prove the correctness of our first static technique:

Theorem 4.1 *Every hybrid consistent run of a program in which all writes are strong and all reads are weak is sequentially consistent.*

Theorem 4.1 is very 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 4.1 only to the synchronization code, and label all other memory accesses as weak. We demonstrate this method for mutual exclusion in the full paper. Given a mutual exclusion algorithm designed for sequentially consistent memories, we produce a modified algorithm by labeling all the writes in the synchronization part of the code as strong, while all other operations are labeled as weak. We prove that the modified algorithm guarantees mutual exclusion (in a strong sense) on hybrid consistent memories.

In the full paper we prove the correctness of our second static technique:

Theorem 4.2 *Every hybrid consistent run of a program in which all reads are strong and all writes are weak is sequentially consistent.*

The proof of this theorem relies on the following assumptions about the run: *(a)* every value written to the same object is unique, and *(b)* every value written is returned by some read. We show by specific counter-examples in the full paper that both these assumptions are necessary to prove this result. The unique writes assumption is often made for proving properties about various consistency conditions [23, 30]; it is sometimes regarded as being merely technical, made in favor of simplicity and having no effect of the correctness of the claims. Our counter-example indicates that this assumption is not merely technical, and that special care should be taken whenever it is made.

5 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 [28, 11, 10]. 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.

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 $op_i^1 \xrightarrow{\alpha} op_j^2$.

[4] In [2, 21, 22], 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.

The relation *happens before*, denoted by $\stackrel{hb}{\Longrightarrow}$, is the transitive closure of the union of $\stackrel{po}{\Longrightarrow}$ and $\stackrel{so}{\Longrightarrow}$.

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

In the full paper, we prove:

Theorem 5.1 *Every hybrid consistent run 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 2.2), 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.

6 Asymmetric Condition

In this section we present a (relatively minor) modification of the definition of hybrid consistency that distinguishes between release and acquire operations. We then modify the definition of a data race to accommodate this distinction. We conclude by proving that asymmetric data-race-free programs behave on asymmetric hybrid consistent memory implementations as if they were sequentially consistent.

We believe our definition of asymmetric hybrid consistency captures a similar semantics to what is captured by the sufficient conditions presented by Adve and Hill [3]. We believe it also captures the main ingredients of the intended semantics of the Stanford Dash multiprocessor [20], as formalized in [22, 21]. Our definition, like the definition in [22, 21], does not include *nsync* operations.[5] See Section 7 for a discussion of similar results that have been proved by others.

We assume it is possible to mark some strong instructions in a program as *releases* and some as *acquires* (an operation can be both a release and an acquire, but this is not necessary). An instance of a release instruction is a *release operation* and an instance of an acquire instruction is an *acquire operation*. Furthermore, there is a *pairing* (a binary relation) defined between release operations and acquire operations. We denote by rop_i a release operation invoked by process p_i and by aop_i an acquire.

The definition of an *asymmetric hybrid consistent* run is the same as that of a hybrid consistent run, except that properties 3 and 4 are replaced with:

3. If $op_j^1 \stackrel{fcs_j}{\Longrightarrow} op_j^2$, and either op_j^1 is an acquire or op_j^2 is a release, then $op_j^1 \xrightarrow{\tau_i} op_j^2$, for any j.
4. If $rop_j^1 \in S$ and $aop_k^2 \in S$ are paired, then $rop_j^1 \xrightarrow{\tau_i} aop_k^2$, for any j and k.

The third property distinguishes between the asymmetric effects of acquire and release operations, while the fourth property guarantees the ordering only for strong operations that are paired.

In the full paper, we define a program to be *asymmetric data-race-free* analogously to the definition of data-race-free. The difference is that the $\stackrel{so}{\Longrightarrow}$ relation is replaced by the $\stackrel{so*}{\Longrightarrow}$ relation, which only orders paired release and acquire operations. The proof of the next theorem follows the proof of Theorem 5.1 very closely.

Theorem 6.1 *Every asymmetric hybrid consistent run of an asymmetric data-race-free program is sequentially consistent.*

[5] More precisely, nsyncs are handled exactly like syncs (strong operations) in [22, 21], a solution that works because of the strong assumptions made in those papers, namely no pipelining of reads [22] and unique sequence numbers on values written [21]. As we pointed out in Section 4, these are significant assumptions.

7 Related Work

Many existing formal treatments of memory consistency conditions [25, 27, 10, 22] 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 [2, 3, 20, 22, 21], and contain many similarities with our work.

These formalisms may be partitioned into two categories: total consistency conditions and partial consistency conditions. Total consistency conditions define the behavior of their implementations for all programs, while partial consistency conditions merely require that certain programs will behave as if the hardware is sequentially consistent. Thus, they are not defined for programs that do not obey a specific condition, even if these programs are sensible. The first category includes release consistency [20, 22, 21], and the sufficient conditions for implementing DRF0 [2] and DRF1 [3]. The second category includes DRF0 [2], DRF1 [3], and PL-programs [22, 21].

Our framework provides total consistency conditions. In Section 4 of the full paper we give examples of programs that are neither DRF0, DRF1, PL-programs nor data-race-free programs. Yet, these programs are correct and efficient under hybrid consistency. Hence, it is interesting to have the consistency conditions be defined for any program.

The major feature distinguishing our approach is the focus on the specification of the conditions in a way that is machine-independent and high-level enough for the programmer. This is the reason our definitions are specified at the interface between the application program and the system. (This is along the lines of [27, 25, 10].) Other formalizations of total consistency conditions [2, 3, 4, 22, 21] focus on hardware implementations that can guarantee certain conditions efficiently.

Defining consistency conditions at the interface between the mcs and the interconnection network, as is done in the definition of weak ordering [16], the definition of release consistency [21, 22, 20], the sufficient conditions for DRF0 [2] and the sufficient conditions for DRF1 [3], has two severe drawbacks. First, this method leads to very detailed and complex definitions and makes it almost impossible to reason directly with the consistency conditions or to argue about the correctness of programs running on implementations of these conditions (see also in [18, page 2]); in contrast, our framework allows one to reason directly with the consistency conditions.[6] The second fault is that the consistency conditions are then bound to the optimizations that are specifically mentioned by the definition. In our framework, the hardware optimizations are not part of the consistency condition, contributing to programs that are more portable and whose correctness is independent of advances in technology.

The work of Adve and Hill deals formally with non-sequential execution of memory operations, although not explicitly. In particular, while the sufficient condition for DRF1 [3, 4] includes a formal treatment of control, this condition is based on the notion of a read operation controlling a write operation by the same processor. This is an operational notion and it is not proven that it captures all the possible ways one operation can control another. In contrast, our approach is syntactic, based on using the actual control instructions in the program, and thus is safer. Most previous work on specifying consistency conditions has either totally ignored control instructions [6, 10, 11, 27, 25] or has merely made the intuitive informal requirement that uniprocessor control dependences are preserved [2, 20, 21, 22].

The work of Gibbons and Merritt [21] deals formally and explicitly with pipelining of memory operations. In their definition, the program is not explicitly modeled, and it is not clear how the intended semantics of the program is preserved. In contrast, our framework explicitly models the program and the run-time system executing it. Unlike our paper, their results do not encompass arbitrary out-of-order or speculative execution of operations. Yet recent experiments [26, 39] have shown that without speculative execution, significant speedup due to parallelism cannot be achieved for programs with complex control flow. Thus it is important to allow for it.

The approach of programming with hybrid consistency by running data-race-free programs was pioneered by Adve and Hill [2], where they focused on implementations. We have applied this approach to the programmer's interface. Similar theorems have been proved in other papers [2, 3, 22, 21], showing that different types of data-race-free programs can be executed on more relaxed implementations of shared memory as if they were sequentially consistent. In our opinion, our result is the only one that combines full support for non-sequential execution, a more natural interface (namely that between the program and the mcs), with being short and comprehensible while still rigorous. We are not aware of any previous technique that, like our static approach, allows labeling of operations without considering all possible executions of the program.

[6] See for example the short and formal proof of correctness for a program that solves the mutual exclusion problem, for any hybrid consistent memory, in [10].

8 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 simplistic 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 environment 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 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.

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.

Acknowledgements: We thank Kourosh Gharachorloo for helpful comments on a previous version.

References

[1] R. Acosta and J. Kjelstrup and H. Torng, "An Instruction Issuing Approach to Enhancing Performance in Multiple Functional Unit Processors," *IEEE Transactions on Computers*, Vol. C-35, No. 9, September 1986, pp. 815–828.

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

[3] S. Adve and M. Hill, *A Unified Formalization of Four Shared-Memory Models*, Computer Sciences Technical Report #1051, University of Wisconsin-Madison, September 1991.

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

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

[6] Y. Afek, G. Brown and M. Merritt, "A Lazy Cache Algorithm," *Proc. 1989 ACM Symposium on Parallel Algorithms and Architectures*, June 1989, pp. 209–222.

[7] M. Ahamad, R. Bazzi, R. John, P. Kohli, and G. Neiger, "The Power of Processor Consistency," these proceedings.

[8] M. Ahamad, J. Burns, P. Hutto, and G. Neiger, "Causal Memory," *Proc. 5th International Workshop on Distributed Algorithms*, Greece, October 1991, pp. 9–30.

[9] H. Attiya, S. Chaudhuri, R. Friedman, and J. Welch, *Shared Memory Consistency Conditions for Non-Sequential Execution: Definitions and Programming Strategies*, Technical Report LPCR 9302, Laboratory for Parallel Computing Research, Department of Computer Science, The Technion, 1993.

[10] H. Attiya and R. Friedman, "A Correctness Condition for High-Performance Multiprocessors," *Proc. 24th ACM Symposium on the Theory Of Computing*, May 1992, pp. 679–690. Also: Technical Report #719, Department of Computer Science, The Technion, May 1992.

[11] H. Attiya and J. Welch, "Sequential Consistency versus Linearizability," *Proc. 3rd ACM Symposium on Parallel Algorithms and Architectures*, July 1991, pp. 304–315. Also: Technical Report #674, Department of Computer Science, The Technion, October 1991.

[12] R. Bisiani, A. Nowatzyk, and M. Ravishankar, "Coherent Shared Memory on a Distributed Memory Machine," *Proc. International Conference on Parallel Processing*, 1989, pp. I-133–141.

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

[14] A. Dinning and E. Schonberg, "An Empirical Comparison of Monitoring Algorithms for Access Anomaly Detection," *Proc. 2nd ACM Symposium on Principles and Practice of Parallel Programming,* March 1990, pp. 1–10.

[15] A. Dinning and E. Schonberg, "Detecting Access Anomalies in Programs with Critical Sections," *Proc. ACM Workshop on Parallel and Distributed Debugging,* May 1991, pp. 85–96.

[16] M. Dubois, C. Scheurich, and F. A. Briggs, "Synchronization, Coherence and Event Ordering in Multiprocessors," *IEEE Computer,* Vol. 21, No. 2, pp. 9–21.

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

[18] K. Gharachorloo, S. Adve, A. Gupta, J. Hennessy and M. Hill, "Programming for Different Memory Consistency Models," *Journal of Parallel and Distributed Computing,* to appear.

[19] K. Gharachorloo, A. Gupta and J. Hennessy, "Performance Evaluation of Memory Consistency Models for Shared-Memory Multiprocessors," *Proc. 4th International Conference on Architectural Support for Programming Languages and Operating Systems,* April 1991, pp. 245–257.

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

[21] P. Gibbons and M. Merritt, "Specifying Non-Blocking Shared Memories," *Proc. 4th ACM Symposium on Parallel Algorithms and Architectures,* July 1992, pp. 306–315.

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

[23] A. Heddaya and H. Sinha, *Coherence, Non-coherence and Local Consistency in Distributed Shared Memory for Parallel Computing,* Technical Report 92–004, Computer Science Department, Boston University, May 1992.

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

[25] M. Herlihy and J. Wing, "Linearizability: A Correctness Condition for Concurrent Objects," *ACM Transactions on Programming Languages and Systems,* Vol. 12, No. 3, July 1990, pp. 463–492.

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

[27] L. Lamport, "How to Make a Multiprocessor Computer that Correctly Executes Multiprocess Programs," *IEEE Transactions on Computers,* Vol. C-28, No. 9, September 1979, pp. 690–691.

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

[29] J. Mellor-Crummey, "On-the-fly Detection of Data Races for Programs with Nested Fork-Join Parallelism," *Supercomputing '91,* pp. 24–33, November 1991.

[30] M. Mizuno, G. Singh, M. Raynal and M. L. Neilsen, "Communication Efficient Distributed Shared Memories," submitted for publication.

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

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

[33] R. Netzer and B. Miller, "What are Race Conditions? Some Issues and Formalizations," *ACM Letters on Programming Languages and Systems,* Vol. 1, No. 1, March 1992.

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

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

[36] D. Shasha and M. Snir, "Correct and Efficient Execution of Parallel Programs that Share Memory," *ACM Transactions on Programming Languages and Systems,* Vol. 10, No. 2, April 1988, pp. 282–312.

[37] B. Smith, *A Massively Parallel Shared Memory Computer,* Invited lecture, 3rd ACM Symposium on Parallel Algorithms and Architectures, July 1991.

[38] J. Smith, "Dynamic Instruction Scheduling and the Astronautics ZS-1," *IEEE Computer,* July 1989, pp. 21–35.

[39] R. N. Zucker and J.-L. Baer, "A Performance Study of Memory Consistency Models," *Proc. 19th International Symposium on Computer Architecture,* May 1992, pp. 2–12.