Using Information from the Programmer to Implement Shared-Memory Optimizations Without Violating Sequential Consistency [*†]

Sarita V. Adve

Department of Electrical and Computer Engineering
Rice University
Houston, Texas 77005-1892
sarita@rice.edu

Rice University ECE Technical Report 9603, March 1996, revised June 1998

Abstract

The memory consistency model of a shared-memory system is a formal specification of the semantics of shared-memory. The most commonly assumed model, sequential consistency, provides simple semantics but is not easily amenable to high performance. This paper focuses on using information from the programmer to determine system optimizations that will not violate sequential consistency. Previous efforts on using this approach have involved an ad hoc process to postulate useful program information, and complex correctness proofs to confirm the optimizations enabled by the information. The ad hoc nature of this process also does not provide insight on possible future optimizations or useful program information.

This paper develops a general framework to derive an explicit relationship (or mapping) between potential optimizations and the corresponding information. The mapping enables an easier exploration of optimizations for future systems. We apply our mapping to determine the information required for optimizations of out-of-order execution, non-atomic execution of writes, and elimination of acknowledgements. We also apply our mapping to common programming constructs for critical sections and producer-consumer interactions. Using known information about such constructs, we determine the optimizations that can be applied to the constructs without violating sequential consistency. For each application of our mapping, we discover optimizations not allowed by previous work in this area. For example, we show that locks can be executed in parallel with certain subsequent memory operations, and common producer signal operations can be reordered and executed non-atomically.

To preserve the illusion of sequential consistency for programmers, we are restricted to using program information pertaining to only sequentially consistent executions of the program. To meet this restriction, we develop a precondition, called the control requirement, that systems must obey. To the best of our knowledge, current systems already obey the control requirement.

1 Introduction

To write a correct shared-memory parallel program, formal semantics for the memory behavior are needed. The memory consistency model provides such semantics. The commonly assumed memory consistency model, *sequential*

[*]Most of this work was performed while the author was a graduate student at the University of Wisconsin-Madison, where she was partly supported by an IBM graduate fellowship. The work at Rice was supported in part by the National Science Foundation under Grant No. CCR-9502500 and CCR-9410457, and the Texas Advanced Technology Program under Grant No. 003604-016 and 003604-025.

[†]This work is a distillation of the author's Ph.D. thesis [1].

consistency [28], ensures that all memory operations in an execution will appear to execute one-at-a-time (or atomically) and memory operations of a single process will appear in program order. While sequential consistency provides an intuitive programming interface, its requirements of program order and atomicity can restrict performance.

Researchers have proposed several approaches to relax the program order and atomicity constraints of sequential consistency. One common technique is to simply relax the consistency model by explicitly allowing out-of-order and non-atomic execution of certain memory operations [14, 16, 17, 22, 35, 36]. Such models provide significant performance gains but require programmers to forgo the simple interface of sequential consistency. Furthermore, several relaxed models exist and often differ from each other in subtle but significant ways [3, 19]. The variety and complexity of these models complicates the task of porting programs across systems that support different models.

This paper focuses on an alternate, programmer-based, approach. This approach relies on obtaining information from the programmer that identifies parts of the program where an optimization (e.g., out-of-order or non-atomic execution of memory operations) will not violate sequential consistency [1, 5, 6, 20, 22, 18]. Based on the information, the system applies the appropriate optimization to the appropriate parts of the program. As long as the information given is correct, programmers can continue to assume sequential consistency, while simultaneously benefiting from performance-enhancing system optimizations (e.g., those allowed by more relaxed consistency models).

We build on a large body of previous work that has used such a programmer-based approach. In particular, several papers have been published to determine the information needed to exploit optimizations of current relaxed consistency models [1, 5, 6, 9, 10, 20, 22, 18, 23, 24, 30]. However, the process of determining such information has been mostly ad hoc, and does not provide insight into information that can be used for other future optimizations. Further, the proofs that the information is correct are fairly complex [6, 7, 9, 10, 22, 23, 24, 30] and/or prohibit common processor behavior (e.g., some work [9, 10, 23, 24, 30] prohibits true speculative execution as implemented in many current and next generation processors).

In contrast to the ad hoc nature of previous applications of the programmer-based approach, this paper describes a framework that reveals a general relationship (or mapping) between system optimizations and program-level information. The mapping enables an exploration of possible optimizations for future systems, without the complexity of previous work.

We illustrate several applications of the mapping that expose optimizations not allowed by previous work on the programmer-based approach. The mapping can be applied in two ways. First, for a given optimization, system designers can apply the mapping to determine the information needed to use the optimization without violating sequential consistency. Second, system designers may examine known information about commonly used program constructs, and apply the mapping to determine new optimizations for those constructs. The second approach simplifies the programmer's work since the required information is often already known. We apply our mapping to optimizations of out-of-order execution, non-atomic writes, and elimination of acknowledgements, and to program constructs used for critical sections and for producer-consumer interactions. Relative to previous work on the programmer-based approach, each application of our mapping is fairly straightforward and exposes new optimizations (as well as all previous optimizations). For example, we show that critical section locks can be executed in parallel with certain subsequent memory operations and certain producer signals can be executed in parallel and non-atomically, without

violating sequential consistency.

The key complexity with the programmer-based approach arises from the requirement that programmers can be expected to provide information only about program behavior on a sequentially consistent system. We address this issue by developing a pre-condition, called the control requirement, that the system must obey. To the best of our knowledge, current systems already obey the control requirement.

While we use our mapping to explore several optimizations, this paper does not seek to promote or evaluate these optimizations. Whether a specific optimization is worth implementing in a system depends on the tradeoffs between the difficulty of getting the required information from the programmer, the complexity of implementing the optimization, and the resulting performance improvement. These tradeoffs vary greatly across different shared-memory implementations, and are particularly difficult to predict for the future.

The rest of the paper is organized as follows. Section 2 describes a specification for sequential consistency. Section 3 uses the specification to develop a mapping between information and optimizations. Section 4 applies the mapping to several optimizations. Section 5 examines common synchronization constructs and applies the mapping to determine the optimizations that can be applied to those constructs. Section 6 describes the constraints a system must obey to use the mapping. Section 7 discusses related work. Section 8 concludes the paper.

2 A Specification for Sequential Consistency

Section 2.1 gives a specification for sequential consistency, assuming atomic memory. Section 2.2 describes four observations that lead to a less restrictive specification. Section 2.3 provides extensions for non-atomic memory. This section combines concepts from several previous works [7, 13, 19, 20, 29, 34], as further elaborated in Section 7.

2.1 A Simple Specification

We first give some definitions and then a specification for sequential consistency.[1] For simplicity, we first consider systems where memory operations occur instantaneously or atomically (but not necessarily in program order). The concept of atomicity is formalized in the definition of an execution order below. Henceforth, the term *operations* refers to memory operations (reads and writes).

Definition 1: *Program order ($\xrightarrow{po}$):* For a given execution, the program text imposes an order on the operations of an individual process. The union of these per-process orders is the *program order*, denoted by $\xrightarrow{po}$.

Definition 2: *Conflicting operations:* Two memory operations *conflict* if they access the same locations and at least one of them is a write [34].

Definition 3: *Execution order:* With atomic memory, there is a total order on all memory operations of an execution, such that a read R returns the value of the last write to the same location that is before R in this total order. This order is called the *execution order* [13].

[1] For brevity, we do not give formal definitions of intuitive concepts such as program, execution, and result of an execution [1]. For simplicity, this paper assumes that for any execution, the number of instruction instances ordered before any instruction instance by program order is finite. This requirement, called *finite speculation* [1], does not prohibit programs that execute infinite instructions; it only restricts speculative execution beyond potentially unbounded loops. A relaxation of this requirement is described in [1].

We say an operation O_1 *executes before (or after)* an operation O_2 if O_1 is before (or after) O_2 by execution order.

Definition 4: *Conflict order ($\xrightarrow{co}$)*: Let X and Y be two operations in an execution. X is ordered before Y by conflict order ($X \xrightarrow{co} Y$) if X and Y conflict and X executes before Y [7].

Definition 5: The *program/conflict graph* for an execution is a directed graph where the vertices are the memory operations of the execution and the edges represent the program order and conflict order on the operations.

Definition 6: An *ordering path* is a path in the program/conflict graph that is between two conflicting memory operations and has at least one program order edge.

Figure 1 illustrates the above definitions. Consider an execution of the program in part (a) where processor P0 executes its writes to X and Y out of program order, processor P1 reads Y after P0 modifies it, and P1 reads X before P0 modifies it. Figure 1(b) illustrates the corresponding execution order. Figure 1(c) illustrates the program order and the corresponding conflict orders and the program/conflict graph. The path Write,X $\xrightarrow{po}$ Write,Y $\xrightarrow{co}$ Read,Y $\xrightarrow{po}$ Read,X constitutes an ordering path of the program/conflict graph in part (c). A specification for sequential consistency follows.

Specification 1: *Specification for sequential consistency (assuming atomicity):* A system is sequentially consistent if the following holds for every execution on the system: if there is an ordering path from memory operation X to memory operation Y, then X executes before Y.

Proof: The specification prevents any cycles in the program/conflict graph. An acyclic graph implies a total order on the memory operations of the execution where the operations of a given process are in program order and a read returns the value of the last conflicting write before it by the above total order. This satisfies the program order and atomicity requirements of sequential consistency. □

Various forms of the above specification have also been proposed by others [13, 29, 34]. Part (c) of Figure 1 illustrates the specification. Consider the ordering path from Write,X to Read,X through the write and read of Y. The specification requires the write of X to occur before the read of X in execution order. This is not true for the execution order of Figure 1 and the execution does not appear sequentially consistent. Imposing the specification would require that the read of X return the value of the write of X and ensures sequential consistency.

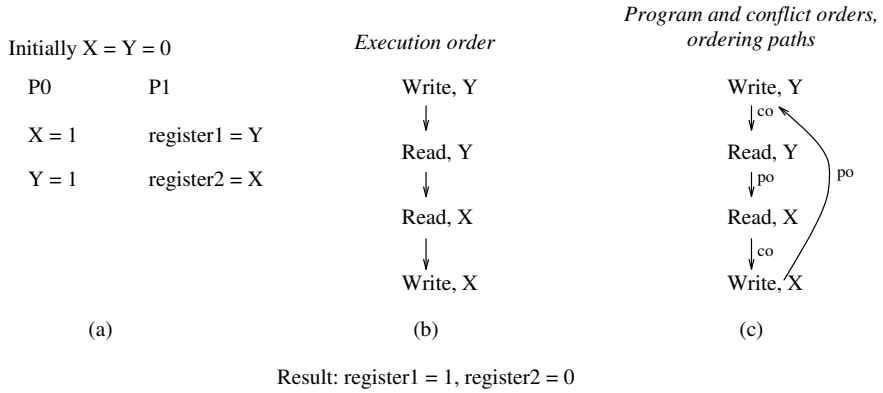


Figure 1 Execution order, program order (po), conflict order (co), and ordering paths of an execution. The depicted execution does not obey Specification 1 for sequential consistency because there is an ordering path from Write,X to Read,X, but Write,X executes after Read,X.

For an atomic system, we say *an ordering path from X to Y executes safely* if X executes before Y; otherwise, we say the *path is violated.* We say an execution of a program is sequentially consistent if it occurs on a sequentially consistent system.

2.2 A Less Restrictive Specification

In practice, Specification 1 can be satisfied by placing several types of restrictions on the execution of memory operations on ordering paths. The simplest way is to ensure that memory operations on a program order edge of an ordering path are executed one-at-a-time in program order. This section describes four observations that lead to a less restrictive specification of sequential consistency, by requiring that restrictions be imposed only on a subset of ordering paths.

Figure 2 illustrates the concepts of this section. The figure shows a program and relevant parts of the program/conflict graph of an execution of the program. For clarity, a few program and conflict order edges are not shown in the figure. All program order edges shown in the figure are on ordering paths, except the edge between the reads of C. The simple technique for ensuring safe execution of ordering paths requires that all memory operations in the figure, except two reads of C, must execute in program order. The following observations allow ignoring some ordering paths and exploiting more parallelism.

The first observation (also made by Shasha and Snir [34] in a different form) is that if there are multiple ordering paths from operation X to operation Y, then the system needs to restrict operations on only one of those ordering paths. For example, in Figure 2(b), there are three ordering paths from Write,A to Read,A:

Write,$A \xrightarrow{po}$ Write,$B \xrightarrow{po}$ Write,$C \xrightarrow{co}$ Read,$C \xrightarrow{po}$ Read,$B \xrightarrow{po}$ Read,A
Write,$A \xrightarrow{po}$ Write,$B \xrightarrow{co}$ Read,$B \xrightarrow{po}$ Read,A
Write,$A \xrightarrow{po}$ Write,$C \xrightarrow{co}$ Read,$C \xrightarrow{po}$ Read,A

Serializing operations on program order edges of all the above paths would require processor P0 to execute its three writes one-at-a-time. Instead, it is sufficient to serialize operations on program order edges of only the third path. This allows P0's writes to A and B to be executed in parallel.

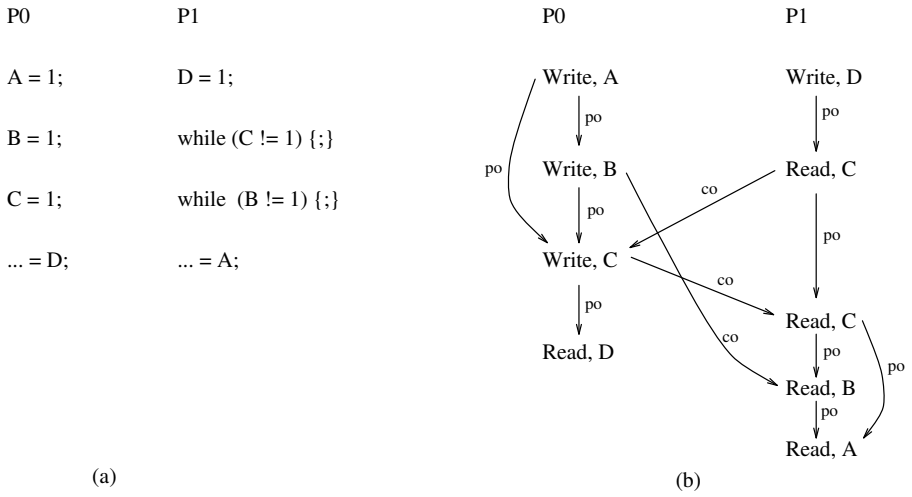


Figure 2 Example for observations 1, 2, and 3. Only one ordering path from Write,A to Read,A needs to be restricted to preserve sequential consistency.

The second observation is that certain operations, called *unessential* operations, can be removed from an execution without affecting its result [20]. Consequently, the system need not restrict ordering paths due to unessential operations. For example, consider the while loops in Figure 2(a). The loops are only used for control; as long as the loop finally terminates and the terminating read returns the correct value, the reads of the non-terminating iterations can be ignored. These non-terminating reads are unessential. In Figure 2(b), the ordering path from the write of D to the read of D results from an unessential read on C; therefore, this path can be ignored and the system need not restrict the operations on this path.

> **Definition 7:** A *synchronization loop* [20] is a sequence of instructions that satisfies the following. (i) The sequence executes one or more reads, called *exit reads*. Depending on whether the value returned by each read belongs to one of specified values, called *exit values*, the construct either terminates or repeats the above. (ii) The loop terminates in every sequentially consistent execution.
>
> An exit read can be part of a read-modify-write if it is the only (static) exit read instruction in the loop.[2]

> **Definition 8:** *Unessential and essential operations:* A set of operations in an execution are *unessential* if they are from an iteration of a synchronization loop that does not terminate the loop. Other operations are *essential* [20].

The third observation pertains to ordering paths between certain writes and essential exit reads. Irrespective of how these paths are executed, the essential exit reads will always execute after these writes (assuming other paths are correctly executed) [20]. Therefore, the system need not restrict these ordering paths. For example, consider the final read of B in the while loop of Figure 2. In Figure 2(b), there is an ordering path from the write of B to the read of B. However, the loop is waiting for the value written by the write; therefore, no matter how the system executes the operations on the ordering path between the write and read of B, the final read of B will always execute after the write. Therefore, no explicit system restrictions are required to impose this particular order.

Figure 3 illustrates the general case. Below, terms such as *before, after, last, between* refer to the ordering by execution order. Let Y be an essential exit read of a synchronization loop. We consider a set of writes that execute before Y and conflict with Y. Let W be the write whose value Y returns (i.e., W is the last conflicting write before Y and writes an exit value for Y). Let W' be the last conflicting write before W that writes an exit value for Y (assuming W' exists). Let $W1$, $W2$, ... be writes that conflict with Y and are between W' and W. $W1, W2, \ldots$ do not write exit values for Y. Suppose we ensure that Y will execute after *one* of $W1$, $W2$, ..., and that the various writes themselves are correctly ordered with respect to other conflicting writes. Then it follows that Y will also execute after W; otherwise, Y would return the (non-exit) value of a write between W' and W and therefore Y would be unessential. It follows that for the set of ordering paths to Y from writes between W' and Y (the dashed paths in Figure 3), only one ordering path needs to be restricted; the rest of the paths in the set can be ignored. If W' does not exist, then no ordering path to Y need be restricted (as in the example of Figure 2).

For the above observation, we also need to model initial values of a location. Henceforth, we assume a hypothetical write to each memory location that writes the initial value of the location at the beginning of the execution order.

The fourth observation uses the notion of consecutive conflicting operations.

[2] More general forms of synchronization loops are possible [1, 20]. A read-modify-write, used in Definition 7, is a read followed by a write to the same location. No other conflicting write can occur between the read and write of a read-modify-write (as ordered by the execution order) [1].

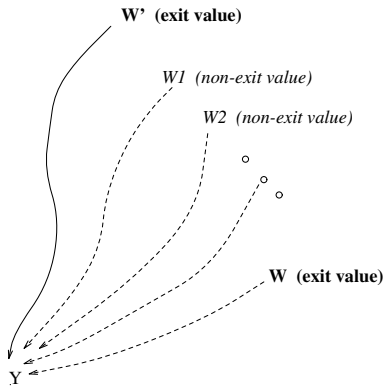


Figure 3 General case for Observation 3. Only one of the dashed ordering paths (if it exists) needs to be restricted.

> **Definition 9:** Two conflicting operations, X and Y, are called *consecutive* if for every ordering path between X and Y, there is no other write on the path that conflicts with X and Y.

It follows that as long as ordering paths between consecutive conflicting operations are executed safely, all ordering paths will be executed safely. This observation does not enable new optimizations, but simplifies our analysis by removing the need to analyze paths between non-consecutive conflicting operations.

Modified Specification. Based on the above four observations, we define a critical set and modify Specification 1. The term *critical* is inspired by Shasha and Snir [34]. An explanation of the various parts of the specification follows below. Again, we use the terms *last, before, after, and between* to refer to the ordering by execution order.

> **Definition 10:** A set of ordering paths of an execution is called a *critical set* if it obeys the following properties. Ignore all unessential operations of the execution. Let Y be any (essential) operation in the execution.
>
> *Case 1:* Y is not an exit read of a synchronization loop: If X and Y are consecutive conflicting operations and if there is an ordering path from X to Y, then one such path is in the critical set.
>
> *Case 2:* Y is an exit read of a synchronization loop: Let W' and W be the last two conflicting writes before Y that write an exit value for Y, and let W' be before W. If W' exists, then consider the set of ordering paths that begin from writes between W' and Y, and that end at Y. If this set contains an ordering path that ends in a program order edge, then one such path is in the critical set.
>
> For every execution, we consider one specific critical set, and call the paths in that set as *critical paths.*

> **Specification 2:** *Specification for sequential consistency (assuming atomicity):* A system is sequentially consistent if the following holds for every execution on the system: if there is a critical path from operation X to operation Y, then X executes before Y.

The formal proof of the above specification is straightforward [1]. Informally, a critical set is defined as the set of ordering paths that are not excluded by the four observations of this section. The definition exploits observation 2 by ignoring all unessential operations. Case 1 in the definition exploits observation 1 by choosing only one ordering path between a pair of conflicting operations to be critical. It exploits observation 4 by considering only consecutive conflicting operations. Observation 3 concerns synchronization loop reads and so is not applicable to case 1. Case 2 of the definition exploits observation 3 by ignoring all ordering paths indicated by that observation. The reason for considering an ordering path that ends in a program order edge is subtle and concerns the later restriction of using information from only sequentially consistent executions [1]. Case 2 exploits observation 4 since it only considers

ordering paths that affect the safe execution of a path between conflicting operations that are consecutive. It exploits observation 1 since for a pair of conflicting operations, at most one path is chosen to be critical.

2.3 Extensions for a Non-Atomic System

Non-atomicity typically occurs in the presence of replication, when different processors may see the effect of a write at different times (e.g., with caches). To discuss non-atomic systems independent of specific hardware configurations, we adopt an abstraction developed by Collier [13]. The abstraction associates each processor with a logical copy of shared memory. A write consists of as many sub-operations as there are processors; each sub-operation atomically updates the memory copy of a unique processor. A read consists of a single atomic sub-operation that returns the current value in the memory copy of the processor that issued the read.

> **Definition 11:** *Conflicting sub-operations:* Two sub-operations conflict if their operations conflict and the sub-operations execute in the same memory copy.
>
> **Definition 12:** *Execution order on sub-operations:* There is a total order on all memory sub-operations of an execution, such that a read sub-operation R returns the value of the last conflicting write sub-operation before R in this total order. This total order is called the *execution order* on sub-operations [13].
>
> We say a sub-operation S_1 *executes before (or after)* sub-operation S_2 if S_1 is before (or after) S_2 by the execution order on sub-operations.

For simplicity, this paper assumes that if two writes conflict, then any sub-operation of one of these writes executes before the corresponding conflicting sub-operation of the other write. This requirement, called *coherence*, is easily met by most hardware cache coherence protocols, and allows a straightforward adaptation of the specification for sequential consistency developed for atomic systems (a relaxation of the coherence requirement appears in [1]):

> **Specification 3:** *Specification for sequential consistency on a non-atomic system:* A system is sequentially consistent if the following holds for every execution on the system: if there is a critical path from operation X to operation Y, then any sub-operation of X executes before the corresponding conflicting sub-operation of Y.

On non-atomic systems, we say an *ordering path from operation X to operation Y executes safely* if each sub-operation of X executes before the corresponding conflicting sub-operation of Y; otherwise, we say *the path is violated*.

3 A Mapping Between Optimizations and Information

We next discuss how information from the programmer can be used to facilitate system optimizations without violating sequential consistency. Call an optimization *safe* if it does not violate sequential consistency. Many optimizations are safe to use for certain parts of the program, but not for others. For example, out-of-order execution is often safe for memory operations on data locations, but is often unsafe for operations used for synchronization [6, 22]. What information can the programmer provide so that the system can determine the operations to which it is safe to apply a given optimization? Similarly, often some information may already be known about the program; e.g., through the use of special synchronization libraries or program constructs provided by the system. How can system designers determine optimizations that would not violate sequential consistency, given the known information?

From Specification 3, *an optimization is safe if it is applied only to the parts of the program that will not result in violating a critical path of the resulting execution.* Therefore, useful information that the programmer could provide to enable safe use of an optimization is to *distinguish parts of the program where the optimization will not violate any critical paths of any execution of the program on the resulting system.*

The above relationship between an optimization and information, however, is not yet useful for our purpose because it requires information on critical paths of *all* executions on a system not yet known to appear sequentially consistent. The primary advantage of the programmer-based approach is that it allows retaining the interface of sequential consistency for programmers; using the above relationship would eliminate this advantage by requiring information about non-sequentially consistent executions. We therefore need a relationship between optimizations and information where the information pertains to critical paths of *only sequentially consistent executions.* To achieve this, we have developed a pre-condition for systems, called the *control requirement*. For systems that obey the control requirement, an optimization is safe if it does not violate critical paths of sequentially consistent executions.

A large part of the complexity in the correctness proofs of previous work on the programmer-based approach arose from the restriction that information from only sequentially consistent executions may be used. This work localizes the above complexity to the specification of the control requirement and its proof of correctness. Section 6 motivates and describes the specification; the proof of correctness appears in [1] and is omitted here for lack of space. The specification and the proof are both fairly complex because we would like to formulate the least restrictive specification possible. To the best of our knowledge, our specification is obeyed by all current systems. Furthermore, the complexity related to the control requirement is a one-time effort; system designers need be concerned most about applying our mapping between optimizations and information which is relatively straightforward.

The following theorem and mapping summarize the above discussion.

> **Theorem 1:** A system is sequentially consistent if for every execution on the system (1) all ordering paths that can be critical paths in some sequentially consistent execution execute safely, and (2) the control requirement described in Section 6 is obeyed.
>
> **Mapping Between Optimizations and Information:**
>
> *Potential optimization:* Any optimization that can violate an ordering path. (Other optimizations do not violate sequential consistency and so do not need information from the programmer.)
>
> *Information that allows safe use of the above optimization:* Identify operations to which the optimization can be applied without violating critical paths of any sequentially consistent execution.

The mapping is deliberately abstract as it is intended to capture a general relationship between optimizations and information, and is not expected to be used directly by programmers. The following sections show how the mapping can be used to specify more concrete information for concrete optimizations.

We note a few aspects of the above mapping; these are common to the information required by previous work on the programmer-based approach as well. First, the mapping does not prohibit any programming styles. Only programmers who wish higher performance need to provide some information about the program, typically by using special constructs for certain operations. This information may be provided incrementally for increasingly higher performance. Second, the mapping requires reasoning about *all* sequentially consistent executions, which may sound complex. However, writing a correct program or designing a correct system in any case requires (at least conceptually) reasoning about all executions possible on the system. Restricting this reasoning to only sequentially consistent

executions actually decreases the number of executions that the programmer or system designer are required to consider. Finally, programmers can typically identify only static operations specified by the program, while determining whether a critical path of an execution is violated requires consideration of dynamic operations of the execution. Thus, the above mapping implicitly requires relating dynamic operations in a sequentially consistent execution to static operations in the program. The following sections illustrate this is easily done.

4 Application of the Mapping to Specific Optimizations

This section illustrates the use of the mapping derived in the previous section by applying it to specific optimizations. We consider the optimizations of executing operations out of program order, non-atomic writes, and eliminating acknowledgements for writes in a cache-based system. In particular, we characterize the operations to which the optimizations can be safely applied, and indicate mechanisms to directly identify such operations to the compiler and hardware. The next section illustrates an easier-to-use application of our mapping, where operations of high-level program constructs are shown to obey several of the characterizations derived in this section.

For the following analyses, when considering critical paths, we implicitly consider critical paths of only sequentially consistent executions, as allowed by Theorem 1.

4.1 Executing Memory Operations Out of Program Order

From Theorem 1, two operations can be executed out of program order if this does not violate a critical path of a sequentially consistent execution. Out-of-order execution of operations that are not on a program order edge of any critical path cannot violate any critical path. It follows that two operations that cannot be on a program order edge of a critical path (of any sequentially consistent execution) can be executed out of program order.

For example, consider the program in Figure 4. We overload the instructions of the program to also represent memory operations of a sequentially consistent execution of the program. A few program and conflict order edges of sequentially consistent executions are shown. The paths between conflicting operations (with at least one program order edge) constitute a critical set of ordering paths. The writes to A, B, C, and D can be reordered with respect to each other because no pair of these writes forms a program order edge of a critical path. Writes of Flag1 and Flag2 can also be reordered for the same reason.

Direct techniques to provide information about program order edges that are never on critical paths are described below. We use the following terminology adapted from message-passing nomenclature. An operation that lies on a conflict order edge of a critical path is called a *communicator*; others are called *non-communicators*. For example, in Figure 4, only operations on Flag1 and Flag2 are communicators. We call the first operation on a conflict order edge a *sender* (e.g., the write of Flag1) and the second operation a *receiver* (e.g., the read of Flag1). If a sender is also on a program order edge, we say the sender *sends* the first operation on that edge. Analogously, if a receiver is also on a program order edge, we say the receiver *receives for* the next operation on that edge. In Figure 4, the write of Flag1 sends the writes of A and B. The read of Flag1 receives for the reads of A and B. Figure 5 shows a canonical critical path, and marks the senders and receivers.

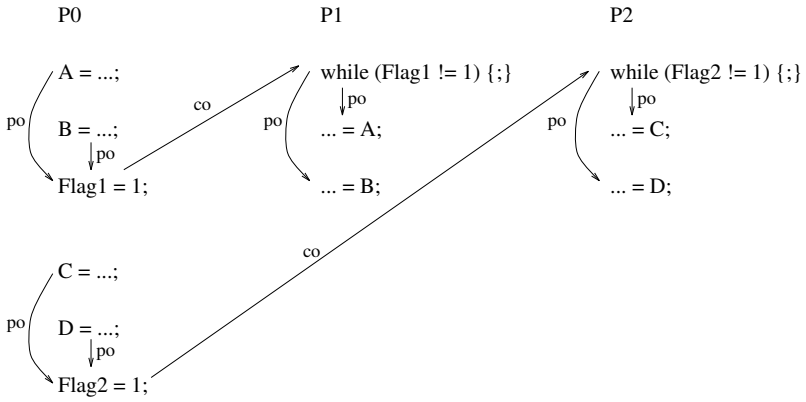


Figure 4 Reordering memory operations. Writes to A, B, C, and D can be reordered with respect to each other without violating sequential consistency since no pair of these writes forms a program order edge of a critical path. Similarly, writes to Flag1 and Flag2 can also be reordered with respect to each other.

The use of message-passing terminology is intentional and suggests an intuitive interpretation of operations on a critical path. In message passing systems, processors communicate by using explicit sends and receives. In shared-memory systems, processors communicate through conflicting accesses to a common memory location. In the absence of additional information, a shared-memory system must assume that any pair of conflicting accesses are implicitly involved in a communication. The conflict order edges of critical paths indicate the true (as opposed to potential) communication in a shared-memory execution; we therefore call the two operations of such an edge as a sender and receiver respectively. Making critical path edges explicit makes the true communication explicit to the system. Note that senders and receivers on the critical path can be reads or writes because any pair of conflicting operations can communicate information in a shared-memory system; further, it is also possible for the same operation to be a sender for some operations and a receiver for other operations.

If op_1 and op_2 are two non-conflicting operations and op_1 precedes op_2 by program order, then op_2 can execute before op_1 if the following holds: op_1 does not receive for op_2 and op_2 does not send op_1 in any sequentially consistent execution. (Program ordered operations that are conflicting need to be ordered since they form an ordering path.)

Direct mechanisms to provide information for out-of-order execution. A general mechanism to distinguish program order edges that may be on a critical path is to associate every operation with the operations it could send and

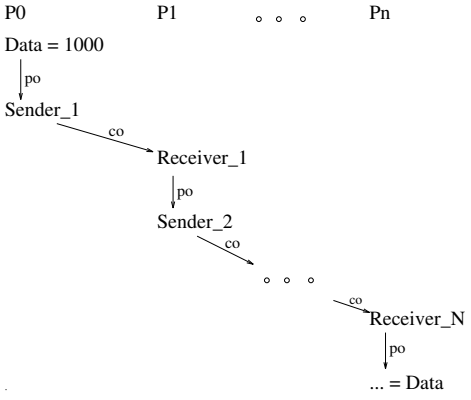


Figure 5 Canonical critical path.

the operations that could receive for it in any sequentially consistent execution. A pair of non-conflicting operations not associated with each other can never be on a critical edge, and can be reordered. The following describes various intermediate mechanisms that are practically easier to implement.

Mechanisms based on distinguishing operations. Earlier work on memory consistency models has used the notion of distinguishing between, or labeling, different types of operations to determine the ordering requirements for various operations [5, 6, 20, 22]. The following describes the operation types that are useful to distinguish to convey useful information within our framework. (Section 7 discusses the relationship of these with mechanisms supported by previous work [5, 6, 20, 22].)

Figure 6(a) specifies several characterizations for when two operations op_1 and op_2 can be reordered, assuming op_1 precedes op_2 by program order and op_1 and op_2 do not conflict. The figure gives four cases depending on whether op_1 and op_2 are communicators or non-communicators. For example, the first entry states that if both op_1 and op_2 are non-communicators, it is always safe for op_1 and op_2 to be reordered. This motivates a mechanism to distinguish non-communicators from communicators. The remaining entries first specify an aggressive characterization requiring mechanisms to associate operations as discussed above, and then the corresponding more conservative characterization requiring mechanisms to distinguish operations. Figure 6(b) summarizes the conservative mechanisms motivated by Figure 6(a). Generic mechanisms to distinguish (or label) operations (e.g., through code annotations or bits in the opcode) and their relative merits are discussed in [1, 3, 18]. The compiler must be able to translate the mechanisms

op_1	op_2	
	Condition when op_1 and op_2 can be reordered when $op_1 \xrightarrow{po} op_2$	
	Non-Communicator	Communicator
Non-Communicator	*Always safe.*	Aggressive: op_2 *does not send* op_1. Conservative: op_2 *does not send any non-communicators.*
Communicator	Aggressive: op_1 *does not receive for* op_2. Conservative: op_1 *does not receive for any non-communicators.*	Aggressive: op_2 *does not send* op_1 *and* op_1 *does not receive for* op_2. Conservative: op_2 *does not send any communicators and* op_1 *does not receive for any communicators.*

(a) Let op_1 and op_2 be non-conflicting operations and let op_1 be before op_2 by program order. The table states when op_2 can be executed before op_1.

Mechanism
Distinguish:
non-communicators
communicators that do not send any non-communicators
communicators that do not receive for any non-communicators
communicators that do not send any communicators
communicators that do not receive for any communicators

(b) Useful mechanisms based on distinguishing operations.

Figure 6 Mechanisms for reordering memory operations.

used at the programming language level to the appropriate mechanisms at the hardware level.

Memory barrier instructions. Several architectures provide special memory barrier instructions to explicitly impose ordering between non-conflicting operations. For example, with the Digital Alpha, program order is maintained between instructions separated by the MB instruction. Thus, a conservative mechanism to associate senders and receivers with the operations they send or receive for is to precede every sender with an MB and follow every receiver with an MB. The mechanism may be conservative in many cases; e.g., *all* operations following an associated sender also effectively get associated with the operations sent by the above sender. The SPARC V9 MEMBAR instruction provides more (but not complete) flexibility in specifying the operations that need to be ordered by the MEMBAR.

The Split-C mechanism. Split-C allows a counter to be associated with every operation [15]. It further defines a sync instruction that is also associated with a counter, and waits for all preceding (by program order) operations associated with the same counter to complete. This provides a mechanism to associate a sender with operations it sends (by preceding the sender with a sync instruction, and associating the operations that the sender sends with the same counter as the sync instruction). As with memory barriers, the mechanism is conservative.

The Tera mechanism. On the Tera machine [8], each instruction is accompanied with a tag that indicates how many subsequent instructions are independent of this instruction. This allows overlapping consecutive independent instructions. On an instruction dependent on an incomplete instruction, Tera switches to a different context. Again, the mechanism is conservative because it does not allow overlapping two independent operations in an instruction stream if there is a dependent operation between them.

Selective acquires. The selective acquires technique [32] induces new (but unnecessary) data dependences between a receiver and any following operations that the receiver receives for [32]. Hardware easily recognizes the induced data dependences, and orders the appropriate operations. Other operations unrelated to the receiver may be inter-mixed with the dependent operations, and may freely execute out-of-order with respect to the receiver. This technique results in some overhead because of the extra instructions used to induce the data dependence; therefore, it cannot be used for all cases.

More aggressive exploitation of reordering information. It is also possible to reorder operations on a program order edge of a critical path since the condition imposed by Theorem 1 is only that the critical path should be executed safely; i.e., the operations at the end-points of the critical path should be executed in the correct order. Referring back to Figure 4, the writes of A and B can be reordered with respect to the write of Flag; the optimization is safe as long as the writes of A and B respectively execute before the reads of A and B. Implementations that exploit this observation are described in [5, 6, 25]. The observation can also be exploited when hardware inherently preserves the order of some (but not all) operations; e.g., operations from a given processor to the same memory module or the same cache line may be guaranteed to execute in the order they are issued [1, 12, 29].

4.2 Non-Atomic Writes

A sequentially consistent system must ensure that writes appear atomic. On systems with caches and a hardware cache coherence protocol, the appearance of atomicity is usually achieved by using the coherence protocol to serialize writes (of all processors) to the same line. Further, a processor is not allowed to read the value of a write until the write is

complete (i.e., until the write is visible to all other processors). For an update-based coherence protocol with a general (non-bus) interconnection network, this typically requires two phases. In the first phase, memory sends updates to caches and receives acknowledgements; in this phase, processors are not allowed to read the updated location since other processors may not yet have seen the update. In the second phase, memory sends messages to the caches indicating they can read the updated value.

We consider relaxing the requirement that the value of a write must be read by other processors only after the write completes; henceforth, we say *a write executes non-atomically* if it employs the above relaxation. The relaxation eliminates the need for the second phase of an update protocol.

To characterize writes that can be executed non-atomically, we need to determine when non-atomic execution can violate a critical path. A simple analysis reveals the following.

> **Lemma 1:** Executing a write W non-atomically can violate a critical path only if either (i) W is a receiver on the path or (ii) the path starts with W followed by a conflict order edge to a read, and the path ends with a read. This assumes operations on a program order edge of the critical path are executed serially in program order.

The proof for the lemma is based on a simple induction on the length of a critical path and uses a straightforward case analysis [1]. Writes that do not satisfy parts (i) and (ii) of the lemma (for every sequentially consistent execution) can be executed non-atomically, and do not require the second phase of an update protocol. In particular, non-communicator writes can be executed non-atomically.

For example, in Figure 4, the writes to A, B, C, and D can be executed non-atomically because they are non-communicators; the writes to Flag1 and Flag2 can be executed non-atomically because they do not satisfy the more general characterization of the writes described in Lemma 1. As another example, consider Figure 7 which shows three processes sharing data through critical sections. To avoid unnecessary lock accesses, P1 and P2 first check (outside the critical section) if some data they want to access has the desired value. These values are updated by P0 and P1 respectively. If the values are not as desired, they do not need to access the lock. Otherwise, they access the lock, but read that data again to ensure they get the latest value. The figure also shows the critical paths. The write of A by

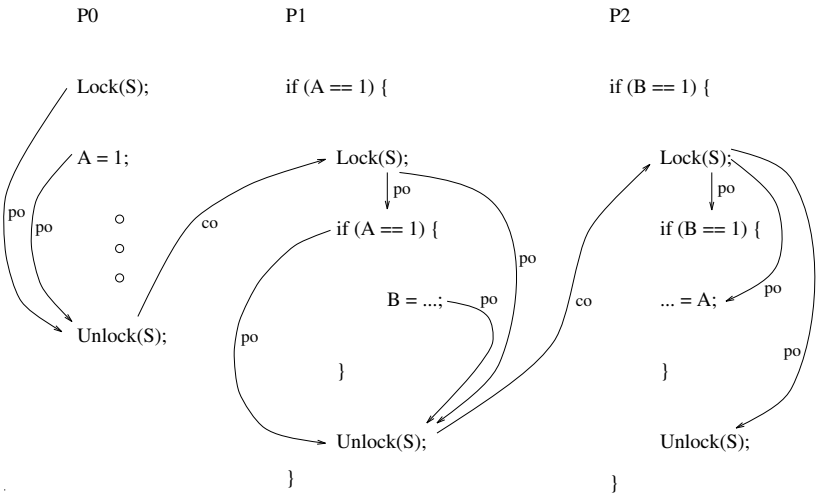


Figure 7 Non-atomic writes. All writes in the figure can be executed non-atomically without violating sequential consistency since they do not satisfy the conditions in Lemma 1.

processor P0 is not a communicator on any critical path; therefore, it does not need two phase updates. Section 5 gives more examples.

A straightforward way to provide information on whether a write can be executed non-atomically is to distinguish writes that do not obey the characterization in Lemma 1 from other writes. Mechanims to make such distinctions are analogous to those for the optimization of out-of-order execution. In particular, writes distinguished as non-communicators can be executed non-atomically.

4.3 Eliminating Acknowledgements

To fulfill the program order and atomicity requirements, common implementations of sequential consistency often require an operation to wait for another operation to complete. For a read, completion is easily detected when the read returns a value. On a system without caches, a write is considered complete when it reaches memory. Typically, with a general (non-bus) interconnection network, memory sends back an acknowledgement to the processor indicating completion of the write. On a system with caches, a write is considered complete when all caches have seen the value written (through invalidations or updates). With a general interconnection network, on receipt of an invalidation or update, a cache usually sends an acknowledgement to the memory or the writing processor. A write is considered complete when all expected acknowledgements are received.

We consider eliminating acknowledgements for some writes. This optimization may be important for bandwidth-limited systems, or for systems that support shared-memory and process acknowledgements in software. Acknowledgements are needed only to indicate the completion of a write operation; such an indication is needed only if some later operation will wait for that write. Thus, to determine when acknowledgements are not required, we need to characterize those writes for which no other operations need to wait for either program order or atomicity. Based on the analysis so far, for program order, no operation need wait for a write if the write is not sent by a sender and if the write does not receive for any operation. For atomicity, no operation need wait for a write if it is safe to execute the write non-atomically as described in Section 4.2. Examples of writes that satisfy the above characteristics are the writes of Flag1 and Flag2 in Figure 4.

Information about a write that does not require acknowledgements can be provided to the system by distinguishing such writes through mechanisms analogous to those for out-of-order execution and non-atomic writes.

5 Using Information About Common Synchronization Constructs

The previous section examined specific optimizations, characterized the operations to which it is safe to apply the optimizations, and indicated direct methods for conveying this information to the system. This section examines common synchronization constructs and shows that many operations of these constructs obey the above characteristics, and can be accordingly optimized. To exploit these optimizations, the only information required of programmers is that they make these constructs explicit to the compiler, which is easily done. We assume there are underlying mechanisms (as discussed in the previous section) for the compiler to convey this high-level information to the hardware, and for the hardware to perform the optimizations possible with this information.

This paper examines typical uses of the lock primitive and producer-consumer interaction. We have also applied our mapping to barriers and commonly used constructs to decrease lock contention [1]. Some of the analysis below may seem lengthy; however, it is shorter and simpler than the proofs in earlier work [6, 7, 22, 23, 24, 30], and only requires analyzing sequentially consistent executions. For the analyses below, any implicit reference to an ordering on operations of an execution (e.g., last, between, before, after) refers to the ordering by execution order.

5.1 Locks and Unlocks

We investigate a typical use of locks and unlocks for implementing critical sections, as described below. We call such locks and unlocks as critical section locks and unlocks, or CS_Locks and CS_Unlocks for short.

In any sequentially consistent execution, CS_Lock and CS_Unlock operations obey the following.

(1) A CS_Lock construct is a synchronization loop whose only shared-memory operation in the final iteration is a read-modify-write that acquires the lock, and the CS_Unlock construct is a write that releases the lock.[3] All CS_Lock constructs accessing the same location have the same exit value.

(2) Locations accessed by a CS_Lock or CS_Unlock are accessed only by other CS_Locks or CS_Unlocks.

(3) A process executes a CS_Unlock only if it was the last process to successfully acquire the lock location (through a CS_Lock) and it has not already issued a CS_Unlock to the same location since its last acquire of the lock.

(4) A CS_Lock operation eventually succeeds in acquiring the lock location.

The above conditions are usually obeyed in the common use of locks for critical sections. Note, however, that programmers are free to use locks in other ways. Such programs will run correctly; only the optimizations discussed in this section will not be applicable to those uses of locks.

We first consider optimizations involving only CS_Lock/CS_Unlock operations. We show that a CS_Lock or CS_Unlock operation can be reordered with respect to a preceding (by program order) CS_Unlock. Further, CS_Lock and CS_Unlock writes can be executed non-atomically and no acknowledgements are needed on a CS_Unlock write. We then consider optimizations involving interactions between CS_Lock/CS_Unlock operations and other operations. We show cases where the entire critical section can be executed in parallel with preceding and/or following (by program order) operations of its process.

5.1.1 Optimizations Involving Only CS_Lock/CS_Unlock Operations

We first derive information about critical paths involving CS_Locks and CS_Unlocks, and then use the information to derive possible optimizations.

Analysis. The following considers only sequentially consistent executions and ignores unessential operations, as allowed by Theorem 1.

`(i)` CS_Lock read is not a sender.

Let X and Y be conflicting operations from CS_Lock or CS_Unlock constructs where X executes before Y. Then either X is a CS_Unlock write and Y is a CS_Lock read, or X and Y are separated by alternating

[3] The general definition of synchronization loops [1] includes constructs such as Test&Test&Set for implementing locks.

CS_Unlock writes and CS_Lock reads. Both cases imply that there is a path from X to Y in the program/conflict graph where the only conflict order edges are from a CS_Unlock write to a CS_Lock read. This implies that if a conflict order edge consisting only of CS_Lock and CS_Unlock operations is on an ordering path, then the edge can always be replaced with another path where only CS_Unlock writes are senders and CS_Lock reads are receivers. When choosing a critical path between two operations, we choose an ordering path where the above replacement has been made. With this choice, the only senders (respectively receivers) from CS_Lock/CS_Unlock constructs are CS_Unlock writes (respectively CS_Lock reads). Therefore, CS_Lock read is not a sender.

`(ii)` CS_Unlock write is not a receiver. (Follows from analysis for `(i)`.)

`(iii)` CS_Lock write is not a sender or receiver. (Follows from analysis for `(i)`).

`(iv)` If there is a critical path from a CS_Unlock write (U) to an operation (O), then U and O must be consecutive conflicting operations and O must be a CS_Lock operation.

From the definition of CS_Locks and CS_Unlocks, O must be either a CS_Lock or a CS_Unlock operation.

First, assume for a contradiction that U and O are not consecutive conflicting operations. The only critical paths between such operations are paths to synchronization loop reads; therefore, O is a CS_Lock read. Further, such a path needs to be critical only if it is from a write that does not write an exit value for the synchronization loop read. However, U writes an exit value for O, a contradiction. Therefore, U and O are consecutive conflicting operations.

Now assume for a contradiction that O is not a CS_Lock operation. Then O must be a CS_Unlock. A CS_Lock write must come between two CS_Unlocks to the same location; therefore, U and O cannot be consecutive conflicting operations, a contradiction.

`(v)` No operation need send a CS_Unlock write (U).

Suppose for a contradiction that some operation sends U. From `(ii)`, U cannot be a receiver. Therefore, the critical path on which U is sent must begin with U. From `(iv)`, the path must end on a CS_Lock operation (say L).

There are two cases depending on whether L is a CS_Lock write or read. If L is a CS_Lock write, then the path $U \xrightarrow{co} R \xrightarrow{po} L$ can be chosen to be critical where R is the read part of L's lock. U is not sent by any operation on this path, proving the proposition.

If L is a CS_Lock read, then L is from a synchronization loop. Therefore, a path to L needs to be critical only if there is a path to L that ends in a program order edge and is either from U or from the last previously executed conflicting CS_Lock write (say W). Either path can be chosen as critical. W must be by U's processor. Therefore, if a critical path is required, then there exists an ordering path from W to L that ends in a program order edge. We can require this path to be critical; therefore, no operation need send U.

`(vi)` No critical path starts with a CS_Unlock write (U) on a conflict order edge to a read (R_1) and ends on a read (R_2).

Suppose for a contradiction that there is a path as described by the above proposition. R_1 and R_2 must be a CS_Lock reads. R_1 is part of a read-modify-write whose write must be between U and R_2. Thus, U and R_2 are not consecutive conflicting operations. This contradicts `(iv)` and proves the proposition.

Optimizations. The above analysis implies that the following optimizations will not violate sequential consistency:

- Two program ordered non-conflicting CS_Unlock writes can be reordered or overlapped (from `(ii)`, `(v)`).
- A CS_Unlock followed by a non-conflicting CS_Lock (by program order) can be reordered or overlapped (from `(i)`, `(ii)`, and `(iii)`).
- CS_Unlock and CS_Lock writes can be executed non-atomically (from `(ii)`, `(iii)`, and `(vi)`).

- No acknowledgements are needed on a CS_Unlock write (from `(ii)`, `(v)`, and since CS_Unlock writes can be executed non-atomically).

Information. To allow the above optimizations, the programming language can support special lock/unlock constructs called `CS_Lock` and `CS_Unlock` respectively. Programmers must use these constructs only if they obey the four constraints mentioned in the beginning of this section for CS_Locks and CS_Unlocks. The constraints describe a common protocol for implementing critical sections. Programmers wanting to use other lock protocols may to continue to do so by using the usual locking constructs provided by the system. The use of the special constructs (as specified) provides enough information to the system to implement the optimizations described in the previous paragraph.

5.1.2 Optimizations Involving CS_Lock/CS_Unlock and Other Operations

From the analysis in the previous section, a CS_Lock read is not a sender, a CS_Unlock write is not a receiver, and a CS_Lock write is neither a sender nor a receiver. It follows that CS_Lock operations can be reordered with respect to preceding (by program order) non-communicator operations of the same process, and CS_Unlock writes and CS_Lock writes can be reordered with respect to the following (by program order) non-communicator operations of the same process.

Further, often, a critical section is used only to ensure that a set of data is updated atomically. This is in contrast to critical sections that are also used to establish some ordering. For example, in the program in Figure 8, locks are used to atomically increment shared locations (A[i]'s) in a critical section. They are not intended to order any operations preceding or following (by program order) their critical sections. Alternatively, the lock does not receive for any operation following (by program order) its critical section and the unlock does not send any operation preceding (by program order) its critical section. It follows that the critical sections in the figure can be executed in parallel with each other, and in parallel with other (non-conflicting) non-communicators preceding and following (by program order) the critical sections.

Task queue based codes are other examples where some of these optimizations are applicable. For example, consider a critical section used to enqueue a task in the task queue. The enqueue lock only receives for the operations in its critical section, and so can be executed in parallel with non-communicators following (by program order) the critical section. Note that the enqueue unlock is typically also used to ensure that the task-specific data produced by the enqueuer is visible to the dequeuer. In this case, the enqueue unlock is a sender for operations preceding (by program order) the enqueue critical section as well, and so cannot be optimized as above. Analogous observations hold for critical sections used to dequeue from a task queue as well.

The above optimizations can be exploited on practical implementations in many ways. Processors that implement non-blocking loads and out-of-order issue provide a natural platform for such optimizations (e.g., through selective acquires [32]). On systems that provide efficient support for computation migration (e.g., through active messages), the computation of the various critical sections of Figure 8 could be migrated to processors that own the corresponding A[i]'s (along with the values of the local[i]'s which are owned by the migrating processor). After the migration is initiated, the migrating processor can proceed with further operations. Specifically, the processor does not have to incur the latency of the lock and data operations in the critical sections; it will need to wait for acknowledgements

```
/* Assume A[i]'s are shared, local[i]'s are local */
...
for (i = my_begin; i < my_end; i++) {
    Lock(L[i]);
    A[i] = A[i] + local[i];
    Unlock(L[i]);
}
...
```

Figure 8 Critical section for atomic updates.

of completion of the critical sections only when it issues a subsequent sender. Finally, software distributed shared memory systems allow more flexibility to exploit the optimization since they can tolerate higher overheads in managing communication; some methods to exploit the optimization on such systems are discussed in [2].

Information. The locks and unlocks in the above examples satisfy the conditions for `CS_Lock` and `CS_Unlock` respectively. The above examples motivate a further distinction depending on whether the communication by the CS_Lock/CS_Unlock operations is restricted only to operations within the critical section. In the general case, we can introduce constructs called `CS_Partial_Lock` or `CS_Partial_Unlock`. These constructs must satisfy all the constraints for `CS_Lock` and `CS_Unlock`. Furthermore, programmers can use `CS_Partial_Lock` if the lock is used to receive only for the operations within the critical section; they can use `CS_Partial_Unlock` if the unlock sends only the operations within the critical section.

Alternatively, the system can provide higher-level software constructs: an `atomic_section` annotation or an `atomic_update`(*location, value*) construct for the critical section of Figure 8, and `enqueue`(*task*) and `dequeue`(*task*) constructs for task queue codes. The programmer can use these constructs to implicitly indicate the necessary semantics for the above optimizations to the compiler. The constructs can be implemented as library routines or can be converted by the compiler into more basic operations recognizable by the hardware. The use of such constructs has been investigated for the Treadmarks system [2]; the above analysis shows why it does not violate sequential consistency.

5.2 Producer-Consumer Synchronization

In a producer-consumer interaction, a producer process typically generates some data and then, using a write to a shared-memory location, signals to the consumer that the data is ready. Call this write a *signal write.* The consumer process meanwhile reads the signal location in a loop, waiting for the value to be written by the signal write. More generally, if the consumer is waiting for multiple producers, the loop may wait for more than one shared-memory location to reach certain values. Call the loop a *wait loop*, its reads as *wait reads*, the locations read by wait reads as *wait locations*, and the values the loop waits for as *exit values*. Figure 9 illustrates an example.

We assume SPMD programs in which every sequentially consistent execution is divided into phases, where each phase ends with a barrier (a degenerate case is where there is only one phase). Further, we require our signal-wait operations to also obey the following reasonable requirements.

In any sequentially consistent execution, signal and wait operations obey the following.

```
                Initially Flag1 = Flag2 = 0
P1              P2                  P3
                                    while ( (Flag1 != 1) || (Flag2 != 1) ) {;}
A = 52;         C = 721;            ... = B;
B = 46;         D = 543;            ... = D;
Flag1 = 1;      Flag2 = 1;          ... = C;
                                    ... = A;
```

Figure 9 Producer-consumer interaction

(1) In any phase, a location specified by a signal or wait can be accessed only by other signals or waits.

(2) If a phase has a signal, then the phase must also have a wait specifying the signal location.

(3) If a phase has a wait, then exactly one (dynamic) signal per wait location must be in the phase.

(4) A phase with a wait construct should begin with each wait location not equal to the corresponding exit value, and a wait loop should eventually terminate.

The following uses the terms signal and wait to imply constructs that obey the properties described above. Programmers are free to use producer-consumer interactions that do not obey the above properties. Such a program will run correctly; only the optimizations considered in this section will not be applicable to those interactions.

We next derive information about critical paths involving signal and wait operations, and then use it to derive the optimizations applicable to the constructs. We show that all combinations of non-conflicting operations from signal and wait constructs, except a wait read followed by a signal write, can be executed out-of-order. Further, signal writes can be executed non-atomically.

Analysis. For two conflicting operations in different phases, there exists an ordering path consisting of only barrier operations (other than the operations being ordered). Assume such a path to be chosen as the critical path. Thus, signal and wait operations are not senders or receivers on any critical path between operations in different phases. Also note that a wait loop is a synchronization loop; therefore, operations from all but the last iteration of such a loop are unessential and are ignored. Again, we consider only sequentially consistent executions, as allowed by Theorem 1.

`(i)` A wait read is not a sender.

For a wait read to be a sender, there must be a write (in the same phase as the read) such that the write conflicts with the read and executes after the read. However, there is only one write to the location of the read in the entire phase and this write writes the exit value for the read. Since the read is essential, it must return the value of the above write. Thus, no conflicting write can execute after the read in the same phase, and the read cannot be a sender.

`(ii)` An ordering path from a write W to a wait read R in the same phase as W need not be critical.

A critical path ending with a wait read R must begin with a signal write. Thus, W is a signal write. Since R is from a synchronization loop and W writes the exit value for R, a path from W to R need not necessarily be chosen as critical. However, we must ensure that if there is a conflicting write W' before W that writes an exit value for R, then some ordering path (to R) from a write between W' and W (and ends in a program order edge) is chosen as critical (if such a path exists). If W' exists, then it must be in a different phase than R. Further, there must be another write that writes a non-exit value for R that is after W' and in a phase before R (since when R's phase begins, the location accessed by R cannot have an exit value.) This write has a path to R consisting of only barriers as senders and receivers, and ending in a program order edge. We choose this path as critical.

`(iii)` A wait read does not receive for another wait read.

Denote the two wait reads as R_1 and R_2 where R_1 is before R_2 by program order (i.e., $R_1 \xrightarrow{po} R_2$). We need to determine if R_1 can receive for R_2. Since R_2 is not a sender (from `(i)`), a critical path with $R_1 \xrightarrow{po} R_2$ must end on R_2, and so it must begin with a write W (in the same phase) that conflicts with R_2. This contradicts our choice of critical paths in `(ii)`.

`(iv)` A signal write is not a receiver.

A signal write can conflict only with a wait read in a phase. Since a wait read cannot be a sender, a signal write cannot be a receiver.

`(v)` No ordering path beginning with a signal write is critical.

An ordering path in a given phase that begins with a signal write must end in a wait read (because a signal write cannot conflict with another write in the same phase). From `(ii)`, the path from a signal write to a wait read is not critical.

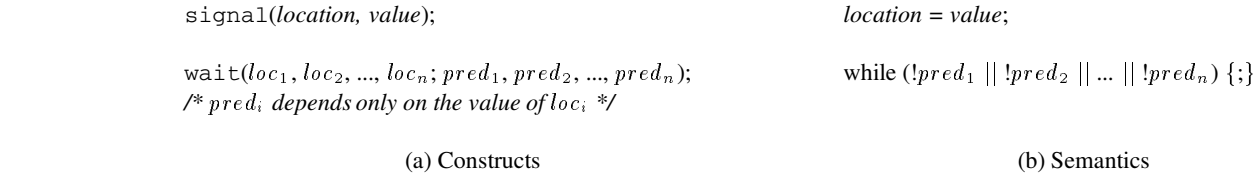
`(vi)` No operation sends a signal write.

For an operation to send a signal write, the signal write must either be a receiver on the critical path or must begin the critical path. From `(iv)` and `(v)` above, neither case is true.

Optimizations. The above analysis implies that the following optimizations will not violate sequential consistency.

- Two program ordered non-conflicting wait reads (from any wait loops) can be reordered (from `(i)` and `(iii)`).
- A signal write followed (by program order) by a non-conflicting wait read can be reordered (from `(i)` and `(iv)`).
- Two non-conflicting signal writes can be reordered (from `(iv)` and `(vi)`).
- A signal write can be executed non-atomically (because from `(iv)` and `(v)`, the write is neither a receiver nor begins a critical path; therefore, from Lemma 1, the optimization is safe.)

Information. The above motivates providing special `signal` and `wait` constructs shown on the left side of Figure 10 with semantics shown on the right side of the figure. Programmers can use the constructs to represent the given semantics only if they obey the constraints (1)–(4) at the beginning of this sub-section. Correct use of the above constructs provides enough information to the system to deduce that the above optimizations are safe. Programmers are free to use other types of producer-consumer interactions, as long as they do not use the special constructs of Figure 10 for those interactions.



(a) Constructs (b) Semantics

Figure 10 Signal-wait constructs

6 System Requirements

This section discusses the control requirement mentioned in Theorem 1 of Section 3. Theorem 1 states that as long as the control requirement is obeyed, the system can violate certain ordering paths without violating sequential consistency. We rely on the programmer to provide information from which the system can deduce which paths to violate. As seen in previous sections, a system can provide several mechanisms to obtain such information. The control requirement is a function of these mechanisms. To give a unified specification of the control requirement for the various possible mechanisms, we use a common abstraction to describe the various mechanisms.

We view the various mechanisms provided by a system as a means to distinguish between two types of ordering paths – paths guaranteed to be executed safely by the system are called the *valid paths* of the system, while other paths are called invalid paths. For example, consider a system that obtains information through a mechanism to distinguish between communicators and non-communicators. The valid paths for such a system are the ordering paths where operations on conflict order edges are distinguished as communicators using the provided mechanism. A characterization of a system in terms of its valid and invalid paths directly represents the optimizations possible on the system. Based on such a characterization, the following re-states the programmer requirements, and the system requirements (from Theorem 1) including a high-level specification for the control requirement.

> **Specification 4:** *Information from the programmer with the valid path abstraction:* All critical paths in a sequentially consistent execution of the program must be distinguished as valid paths for the system. (Equivalently, for any sequentially consistent execution of the program, critical paths of the execution are not distinguished as invalid.) A program that obeys the above constraint is called a *valid program*.

The above specification does not restrict any programming styles. For example, if the system default is that all paths are considered valid, then any program can be trivially made valid without extra effort by the programmer. Programmers who would like higher performance can then distinguish some (non-critical) paths as invalid; further, this can be done incrementally for increasingly higher performance. Also note that the above specification is quite abstract because it is intended to cover a wide range of optimizations; for real systems with specific optimizations, we expect the information about valid and invalid paths to be provided with easier-to-use constructs as described earlier (e.g., the `CS_Lock`, `CS_Unlock`, `signal`, and `wait` constructs).

> **Specification 5:** A system appears sequentially consistent to a valid program if it obeys the following for all executions of the program on the system.
>
> *Valid path requirement:* If there is a valid path from operation X to a conflicting operation Y, then any sub-operation of X executes before the corresponding conflicting sub-operation of Y.
>
> *Control requirement:* The valid paths of the execution form a critical set of the execution.

The proof of correctness of the above specification follows directly from Specification 3 [1]. The following describes examples to motivate the need for the control requirement, and then describes practical, but relatively conservative methods to implement the above high-level form of the requirement. The proof that the practical implementations satisfy the high-level requirement [1] is the most complex part of this work; it is omitted here for lack of space.

Motivation for the control requirement. The control requirement is needed to prohibit situations where the programmer ensures that critical paths of every sequentially consistent execution of the program are valid, the system executes

all valid paths safely, and yet the resulting execution is not sequentially consistent because some critical paths of the execution are not valid. This is possible with a valid program since a valid program guarantees that critical paths will be valid only for sequentially consistent executions. Figure 11 [1, 7] provides examples to illustrate the above situation.

Consider the example in Figure 11(a). In any sequentially consistent execution, the reads of X and Y should return the value 0, and the writes should not be executed. Thus, there are no ordering paths in any sequentially consistent execution and the program is a valid program for all systems. Without the control requirement, it is possible to have an execution where both processors return the value 1 for their reads and execute their writes. In this execution, the ordering path Read,X $\xrightarrow{po}$ Write,Y $\xrightarrow{co}$ Read,Y $\xrightarrow{po}$ Write,X is not executed safely and the execution is not sequentially consistent. The above path is not a critical path of any sequentially consistent execution; therefore, it does not have to be a valid path for the program to be valid. It follows that violating the above path does not violate the valid path requirement of Specification 5. The above path, however, is a critical path of the described execution. The control requirement prohibits the above path from occurring in the execution by requiring that critical paths of an execution be valid paths of some sequentially consistent execution.

Figure 11(b) illustrates another, less straightforward, example. In any sequentially consistent execution of the program, P1's read of Y always returns the value 1; therefore, P1 never issues its write of X. Thus, the only ordering path in any sequentially consistent execution is Write,Y $\xrightarrow{po}$ Write,*Flag* $\xrightarrow{co}$ Read,*Flag* $\xrightarrow{po}$ Read,Y. Assume a system where only the above path is valid. In the absence of the control requirement, an aggressive implementation could allow P0 to write *Flag* before its read of X returned a value. This could result in the following sequence of events which makes P1's read of Y return 0, and violates sequential consistency without violating the valid path requirement: (i) P0 writes *Flag*, (ii) P1's read of *Flag* returns 1, (iii) P1's read of Y returns 0, (iv) P1 executes its write of X, (v) P0's read of X returns 1, (vi) P0 does not issue its write of Y. This execution is not sequentially consistent and it has an ordering path that is not executed safely: Read,X $\xrightarrow{po}$ Write,*Flag* $\xrightarrow{co}$ Read,*Flag* $\xrightarrow{po}$ Write,X. The above path is not a critical path for any sequentially consistent execution, but is a critical path for the above execution. The valid path requirement does not require the safe execution of such a path. The control requirement is needed to prohibit the above path from occurring in an execution.

In general, in the absence of the control requirement, anomalies of the type in Figure 11 may occur because the system is free to execute a logically future operation that results in some non-sequentially consistent behavior, which destroys a future valid path whose safe execution was to have prevented this very behavior. The control requirement breaks the above cycle in causality.

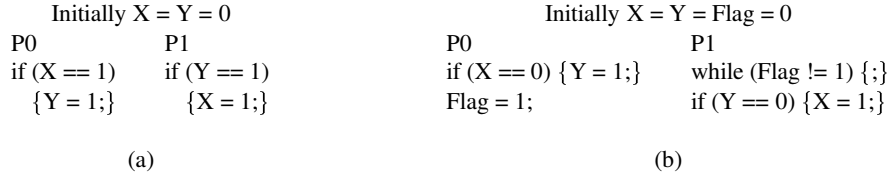


Figure 11 Motivation for control requirement

Practical hardware for the control requirement. The following describes practical but conservative methods to implement the control requirement.[4] Most systems obey one of the conservative methods below. We have also formalized more aggressive methods that are more tightly coupled to the information used and optimizations provided by the system [1], but do not discuss them here because of their complexity.

The most conservative method is to stall a processor on a read until the read returns its value. A more aggressive method allows a processor to proceed with pending reads as long as it does not execute a write sub-operation until (i) it is resolved that the dynamic instruction for the write will be committed with the given address and value (i.e., it will not be squashed), and (ii) it is known which memory operations preceding the write by program order will be executed, which addresses such operations will access, and which values such writes will write. Thus, reads can always be executed speculatively, but writes need to wait until the control flow for the execution and the addresses and values to be accessed by different operations preceding (by program order) and including the write are resolved.

7 Related Work

There is a substantial body of work related to this paper. Due to space constraints, this section covers only systems and optimizations that have been related to sequential consistency. A more comprehensive coverage appears in [1].

7.1 Work Motivated by Hardware and Runtime System Optimizations

Several researchers have proposed relaxed memory consistency models that explicitly specify relaxations of the program order and atomicity requirements of sequential consistency (e.g., [17, 22, 25, 36, 35, 16, 11, 14]). A disadvantage of these models is that their system-centric nature results in several different and complex interfaces for the programmer [3, 19], thereby complicating programmability and portability.

To address the above disadvantages of the system-centric relaxed models, researchers have proposed a more programmer-centric view to describe relaxed models. This view describes the consistency model in terms of information that the programmer must provide so that the optimizations allowed by the model can be safely applied without violating sequential consistency [5, 6, 20, 22]. This paper has focused on the programmer-centric view, and has evolved from previous work on the data-race-free (DRF0, DRF1) and properly-labeled (PL, PLpc) programmer-centric models [5, 6, 20, 22]. The above work, however, uses fairly ad hoc methods to determine appropriate optimizations and programmer information for the safe execution of the considered optimizations. Following the above work, several researchers have proposed various formalisms and developed proofs to show that the information described by the above work will not violate sequential consistency for the considered optimizations [10, 9, 23, 24, 30]. These proofs are fairly complex. To alleviate the complexity, some proofs are restrictive because they implicitly or explicitly assume a strict control requirement. For example, many previous proofs prohibit speculative execution [9, 10, 23, 24, 30], as allowed by many recent processors. Some work has proposed new optimizations and information to use these

[4] The conservative methods were first described in our previous work on programmer-centric models related to optimizations of current relaxed consistency models [4, 7]; this work generalizes their use to cover several more optimizations.

without violating sequential consistency, but has not provided any proof of correctness nor considered any type of control requirement (e.g., [12, 11]).

In contrast to the above work on programmer-centric models, this paper develops a common framework to determine the mapping between information and safe optimizations, and develops a common proof of correctness of a generic system requirement for systems based on the above mapping. The mapping reveals optimizations that were not allowed by earlier programmer-centric models (as well as all optimizations allowed by previous programmer-centric models), and for which the proofs of correctness are now straightforward. Finally, our system requirements allow fairly aggressive speculative execution, and to the best of our knowledge, do not prohibit current system optimizations.

The specific aspects of our framework that have directly evolved from the previous programmer-centric work are as follows. The notion of synchronization loops is similar to, but slightly more general than, that used for the PLpc model [20]. In particular, a synchronization loop in this work is not restricted to a single exit read per loop as in PLpc, thereby allowing the optimizations for the signal-wait construct in Section 5.2. The system requirements in Section 6 are similar in form to those for data-race-free-1 [7] and PLpc [4]. The key difference is that this work makes explicit the relationship between the optimizations exploited by the system and the information it uses from the programmer. Thus, the system requirements as stated in this work expose optimizations other than those allowed by data-race-free-1 and PLpc, and allow system designers to directly determine the necessary characterizations of operations to which the optimizations can be applied safely.

Many optimizations and information analogous to that described in this paper are also supported by previous programmer-centric models. Figure 12 duplicates the various types of useful information illustrated in Figure 6, and also shows the analogous information (if any) exploited by the previous data-race-free and properly-labeled models. For most cases, the operation types characterized by the earlier models are easier to identify for programmers than the analogous characterizations of this work. For example, data-race-free-1/PL characterize a data (or ordinary or competing) operation as one that is not involved in a race in any sequentially consistent execution. This characterization of a data operation is an easy-to-recognize form of non-communicators. We chose to use the characterization of non-communicators in this work because it is less restrictive; e.g., the write of A by processor P0 in Figure 7 is involved in a race (and therefore cannot be classified as data with data-race-free-1/PL); however, in our framework, the write is a non-communicator. Less restrictive characterizations imply that system designers can potentially apply optimizations to more operations. Once useful optimizations have been ascertained, the system designer can determine appropriate higher level information that would be easiest to obtain from the programmer (e.g., the constructs of Section 5).

Referring back to Figure 12, the PLpc model is the most aggressive of the earlier programmer-centric models since it exploits more information than the others. However, it supports the last two mechanisms in Figure 12 in a limited way. For example, PLpc distinguishes communicators which do not send other communicators from those that do; however, it does so only for read communicators. In Figure 4, the write to Flag1 is a write communicator that does not send other communicators. PLpc does not provide a mechanism for the programmer to indicate this information. Therefore, PLpc systems do not have the information to determine that the writes of Flag1 and Flag2 in Figure 4 can be reordered. Similarly, PLpc does not support information to reorder non-conflicting CS_Unlock writes (Section 5.1.1) or non-conflicting signal writes (Section 5.2) or signal reads (Section 5.2). The reordering of (certain) critical section

locks with data operations following (by program order) the critical section, and of (certain) critical section unlocks with data operations preceding (by program order) the critical section (Section 5.1.2) is also not exploited by PLpc. Finally, the signal writes (Section 5.2) and the write of A in Figure 2.3 are not allowed to be non-atomic in PLpc, and the optimization of not requiring acknowledgements is not considered by the data-race-free and the properly-labeled models.

Gibbons and Merritt have relaxed the system-centric model of release consistency by allowing a release to be associated with a subset of the preceding (by program order) operations [23]. They show that it is sufficient for a release (a sender in our terminology) to wait for completion of only the preceding (by program order) data operations that are associated with it, preceding synchronization writes, and preceding synchronization reads that return the value of a synchronization write of another processor. Section 4.1 shows how to generalize the above observation by allowing a receiver (an acquire in the terminology of [23]) to be associated with operations it receives for as well, thereby allowing more optimizations with acquires. Further, we also allow a sender to not wait for the preceding synchronization operations that are not associated with it. Finally, Gibbons and Merritt do not state any control requirement; as stated, their conditions seem to allow the anomalous behavior described for the program in Figure 11.

For all the optimizations, our approach of considering high-level program constructs and obtaining information implicitly in terms of how such constructs should be used leads to easier-to-use systems (even if for some cases, the considered optimizations are already allowed by previous models). Our work also provides a unified approach to reasoning about how all of the above optimizations can be applied without violating sequential consistency.

Jointly with others, we have also proposed a framework for uniformly specifying previous system-centric models and system constraints for future models [19]. That framework is closely related to our method for specifying system requirements in Section 6. The system-centric framework proposes that most of the system specification should give the ordering paths that are executed safely by the system. However, unlike the work in this paper, the system-centric work is concerned only with specifying systems in the most aggressive manner. This paper tries to determine the relationship between the ordering paths executed safely by the system and information from the program that will make such a system appear sequentially consistent, and to determine new optimizations based on this relationship.

We have directly used Collier's abstraction for describing shared-memory systems with non-atomic memory [13],

Mechanisms from Figure 6	**Analogous Mechanism in**		
	Data-Race-Free-0	**Data-Race-Free-1/PL**	**PLpc**
non-communicators	data	data/ordinary/ non-competing	non-competing
communicators that do not send any non-communicators	-	acquire synchronization, unpaired sync/nsync	competing read
communicators that do not receive for any non-communicators	-	release synchronization, unpaired sync/nsync	competing write
communicators that do not send any communicators	-	-	loop read
communicators that do not receive for any communicators	-	-	loop write

Figure 12 Mechanisms supported by previous models.

as described in Section 2.3. Using his abstraction, Collier defined various shared-memory architectures as sets of rules, where each rule is a restriction on the order of execution of certain sub-operations. Collier also showed how to formally prove whether a set of architecture rules is stronger, weaker, or equivalent to another set of rules. We have used some of Collier's proof techniques directly in our work [1]. Further, Specification 1 for sequential consistency is a straightforward reformulation of one set of rules proposed by Collier. The rest of our work, however, focuses on providing the illusion of sequential consistency in the presence of various optimizations using information from the programmer; Collier does not consider this issue.

Landin et al. use a specification for sequential consistency similar to Specification 1 in Section 2.1 [29].

Finally, researchers have also proposed hardware techniques to relax program order and write atomicity without violating sequential consistency and without information from the programmer [7, 13, 21, 29]. Hardware prefetching and speculative execution [21] appear to be the most promising of these techniques. A recent study has shown that these techniques can almost eliminate the hardware performance difference between popular relaxed consistency models for some, but not all, applications [33].

7.2 Work Motivated by Compiler Optimizations

Shasha and Snir developed a pioneering compiler algorithm to allow optimizations without violating sequential consistency [34]. Subsequently, this algorithm was modified by Midkiff and Padua to better handle loops [31] and by Krishnamurthy and Yelick to achieve polynomial time complexity in the number of processors [26].

The algorithm developed by Shasha and Snir uses the program order relation P, and a conflict relation C which orders every pair of conflicting operations both before and after each other. The algorithm finds a "minimal" set of cycles in the graph of $P \cup C$. Such cycles are called critical cycles and the operations on their P edges are called critical pairs. Shasha and Snir prove that program order need only be imposed on critical pairs [34].

Our Specification 1 and the reasoning for observations 1 and 4 in Section 2.2 are similar to that used by Shasha and Snir. However, their work (and other subsequent modifications [31, 26]) cannot be directly applied to our context because it requires reasoning about interactions that do not occur on sequentially consistent systems. Specifically, the $P \cup C$ graph orders conflicting pairs of operations both before and after each other. (In contrast, the analogous program/conflict graph of our work orders conflicting operation pairs in a single direction based on the actual order in an execution, and we only consider sequentially consistent executions.) Thus, the algorithm by Shasha and Snir considers several execution paths that can never occur on any sequentially consistent system. It is acceptable for compilers to analyze such paths. However, we cannot expect the programmer to reason about non-sequentially consistent executions; such reasoning defeats our primary goal of providing a sequentially consistent interface in the presence of system optimizations. We overcome the above problem by imposing the control requirement, which allows restricting the analysis to only sequentially consistent executions.

Other differences between the work by Shasha and Snir and this work include the notion of synchronization loops, which allows eliminating further critical cycles as described by observations 2 and 3 of Section 2.1. Shasha and Snir consider incorporating some synchronization information for cases where a pair of synchronization operations executes in a fixed order in all executions; they do not indicate how the compiler can detect such cases. Finally, the

performance of the compiler algorithms depends on the accuracy with which addresses of different memory operations can be disambiguated (i.e., the accuracy of the conflict relation).

Krishnamurthy and Yelick subsequently performed a more substantial modification to the above algorithm to exploit knowledge about synchronization to eliminate certain spurious critical cycles [27]. Similar to the work on programmer-centric models, their modification requires programmers to use explicit synchronization constructs recognizable by the system, and makes certain assumptions about the behavior of synchronization operations (e.g., only one post per event variable for post-wait synchronization). Their work does not explicitly discuss the issue of whether the requirements pertain to sequentially consistent executions or all executions; however, the proofs assume all (and not just sequentially consistent) executions obey the synchronization assumptions. Therefore, again, this work cannot be directly applied to the programmer-based approach.

Finally, we also consider optimizations not considered by the compiler-related work (e.g., non-atomic writes and eliminating acknowledgements in cache-based systems).

8 Conclusions and Discussion

The widening gap between processor and memory speeds increases the need to tolerate memory latency in shared-memory systems. One method is to selectively allow out-of-order and non-atomic execution of certain memory operations. Recent processors have become fairly aggressive in their support to exploit such optimizations (e.g., superscalar and out-of-order issue). Further, a recent trend towards managing shared-memory actions in software can make more complex schemes for selectively applying various optimizations both possible and profitable.

Building on a large body of previous work, this paper considers the approach of using information from the programmer to allow optimizations (related to out-of-order and non-atomic execution of memory operations) without forsaking sequential consistency. Previous work in this area has determined the necessary information for optimizations employed by current relaxed consistency models. However, the process of determining the information has been fairly ad hoc and does not give insight for useful information for other future optimizations. Furthermore, the correctness proofs involved are fairly complex and/or prohibit common processor behavior (e.g., speculative execution of reads).

This paper develops a general mapping that exposes the relationship between an optimization and information the programmer can provide to enable use of the optimization without violating sequential consistency. We have applied our mapping to the optimizations of out-of-order execution, non-atomic execution of writes, and elimination of acknowledgements, characterizing the operations on which these optimizations can be applied without violating sequential consistency. Information identifying such operations can be directly communicated to the compiler and hardware. An alternative application of our characterizations is to exploit known information about commonly used program constructs. We showed that the above optimizations can be applied (without violating sequential consistency) to many of the operations used to implement critical sections and common producer-consumer interactions. For the programmer, this use of our mapping results in a simple method to give the necessary information to the system.

A key feature of the information we expect from the programmer is that it only requires reasoning about sequen-

tially consistent executions. This feature is essential to retain the sequentially consistent interface for the programmer. It is convenient for system designers as well, since they can determine useful information and optimizations by only considering sequentially consistent executions rather than all possible executions. To provide the above feature, we developed a pre-condition for the system called the control requirement, which is fortunately obeyed by most systems. Previous work has often ignored the control requirement or used a requirement not met by several current systems.

Our work does not prohibit any shared-memory parallel programming styles. Programmers can continue to write programs as before. Only when seeking higher performance do programmers need to provide information to the system; this information can be incrementally provided for increasing performance.

There are two main limitations of this work. First, it does not provide an analysis of the benefits of the additional optimizations considered. Such an analysis, however, depends on several factors, some of which may change with time. For example, recent software-based shared-memory systems benefit from optimizations (e.g., lazy release consistency) that would be considered impractical for hardware-based systems. This paper has seeked to develop a general theoretical framework within which optimizations for future systems can be considered, without imposing unnecessary constraints.

Second, this work may be criticized as being too complex. However, we maintain that the subject is inherently complex as demonstrated by other formal treatments of the subject. Compared to previous formal treatments, the analysis for each of the optimizations and program constructs explored in this work is much simpler and more intuitive. The major complexity of this work is isolated in the proof of Theorem 1 (mainly the control requirement), which is a one-time effort. In the future, it may be possible, however, to eliminate more ordering paths from the critical set, thereby allowing more optimizations. In that case, system designers may have to confront the complexity of our proof to determine whether our system specification holds for the new critical set.

Acknowledgements

This work was done as part of my dissertation. I am indebted to my advisor, Mark Hill, for his guidance throughout my dissertation research. I also thank Kourosh Gharachorloo for many discussions that led to some of the ideas in this paper. Kourosh Gharachorloo and Mark Hill also provided valuable comments on drafts of this paper.

References

[1] S. V. Adve. *Designing Memory Consistency Models for Shared-Memory Multiprocessors*. PhD thesis, Computer Sciences Department, University of Wisconsin-Madison, December 1993. Available as Technical Report #1198.

[2] S. V. Adve, A. L. Cox, S. Dwarkadas, and W. Zwaenepoel. Replacing Locks by Higher-Level Primitives. Presented at the *4th Workshop on Scalable Shared-Memory Multiprocessors*, Chicago, May 1994. Available as Department of Computer Science Technical Report TR94-237, Rice University, 1994.

[3] S. V. Adve and K. Gharachorloo. Shared Memory Consistency Models: A Tutorial. *IEEE Computer, Special Issue on Shared-Memory Multiprocessing*, pages 66–76, December 1996.

[4] S. V. Adve, K. Gharachorloo, A. Gupta, J. L. Hennessy, and M. D. Hill. Sufficient System Requirements for Supporting the PLpc Memory Model. Computer Sciences Technical Report #1200, University of Wisconsin-Madison, and Technical Report #CSL-TR-93-595, Stanford University, December 1993.

[5] S. V. Adve and M. D. Hill. A Unified Formalization of Four Shared-Memory Models. *IEEE Trans. on Parallel and Distributed Systems*, 4(6):613–624, June 1993.

[6] S. V. Adve and M. D. Hill. Weak Ordering - A New Definition. In *Proc. 17th Ann. Intl. Symp. on Computer Architecture*, pages 2–14, May 1990.

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

[8] R. Alverson, D. Callahan, D. Cummings, B. Koblenz, A. Porterfield, and B. Smith. The Tera Computer System. In *Proc. Intl. Conf. on Supercomputing*, pages 1–6, Amsterdam, June 1990.

[9] H. Attiya, S. Chaudhuri, R. Friedman, and J. Welch. Shared Memory Consistency Conditions for Non-Sequential Execution: Definitions and Programming Strategies. In *Proc. Symp. on Parallel Algorithms and Architectures*, 1993.

[10] H. Attiya and R. Friedman. A Correctness Condition for High-Performance Multiprocessors. In *Proc. Symp. on Theory of Computing*, pages 679–690, 1992.

[11] B. N. Bershad and M. J. Zekauskas. Midway: Shared memory parallel programming with entry consistency for distributed memory multiprocessors. Technical Report Technical Report CMU-CS-91-170, Carnegie Mellon University, September 1991.

[12] M. Carlton. Implementation Issues for Multiprocessor Consistency Models. In *Presented at the 2nd Workshop on Scalable Shared-Memory Multiprocessors*, Toronto, May 1991.

[13] W. W. Collier. *Reasoning about Parallel Architectures*. Prentice-Hall, Englewood Cliffs, New Jersey, 1992. Parts of this work originally appeared as IBM technical reports in 1984 and 1985.

[14] F. Corella, J. M. Stone, and C. M. Barton. A Formal Specification of the PowerPC Shared Memory Architecture. Technical Report RC 18638(81566), IBM Research Division, T.J. Watson Research Center, January 1993.

[15] D. E. Culler, A. Dusseau, S. C. Golstein, A. Krishnamurthy, T. von Eicken, and K. Yelick. Parallel Programming in Split-C. In *Proc. Supercomputing '93*, pages 262–273, Portland, November 1993.

[16] Digital Equipment Corporation. *Alpha Architecture Reference Manual*, 1992.

[17] M. Dubois, C. Scheurich, and F. A. Briggs. Memory Access Buffering in Multiprocessors. In *Proc. 13th Ann. Intl. Symp. on Computer Architecture*, pages 434–442, June 1986.

[18] K. Gharachorloo. *Memory Consistency Models for Shared Memory Multiprocessors*. PhD thesis, Stanford University, 1995.

[19] K. Gharachorloo, S. V. Adve, A. Gupta, J. L. Hennessy, and M. D. Hill. Specifying System Requirements for Memory Consistency Models. Technical Report #CSL-TR-93-594, Stanford University and Computer Sciences Technical Report #1199, University of Wisconsin-Madison, December 1993.

[20] K. Gharachorloo, S. V. Adve, A. Gupta, J. L. Hennessy, and M. D. Hill. Programming for Different Memory Consistency Models. *Journal of Parallel and Distributed Computing*, 15(4):399–407, August 1992.

[21] K. Gharachorloo, A. Gupta, and J. Hennessy. Two Techniques to Enhance the Performance of Memory Consistency Models. In *Proc. Intl. Conf. on Parallel Processing*, pages I355–I364, 1991.

[22] 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. 17th Ann. Intl. Symp. on Computer Architecture*, pages 15–26, May 1990.

[23] P. B. Gibbons and M. Merritt. Specifying Nonblocking Shared Memories. In *Proc. Symp. on Parallel Algorithms and Architectures*, pages 306–315, 1992.

[24] P. B. Gibbons, M. Merritt, and K. Gharachorloo. Proving Sequential Consistency of High-Performance Shared Memories. In *Proc. Symp. on Parallel Algorithms and Architectures*, pages 292–303, July 1991.

[25] P. Keleher, A. L. Cox, and W. Zwaenepoel. Lazy Release Consistency for Software Distributed Shared Memory. In *Proc. 19th Ann. Intl. Symp. on Computer Architecture*, pages 13–21, 1992.

[26] A. Krishnamurthy and K. Yelick. Optimizing Parallel SPMD Programs. In *Proc. 7th Workshop on Languages and Compilers for Parallel Computing*, 1994.

[27] A. Krishnamurthy and K. Yelick. Optimizing Parallel Programs with Explicit Synchronization. In *Proc. SIGPLAN Conf. on Programming Language Design and Implementation*, July 1995.

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

[29] A. Landin, E. Hagersten, and S. Haridi. Race-Free Interconnection Networks and Multiprocessor Consistency. In *Proc. 18th Ann. Intl. Symp. on Computer Architecture*, pages 106–115, May 1991.

[30] D. H. Linder and J. C. Harden. Access Graphs: A Model for Investigating Memory Consistency. *IEEE Trans. on Parallel and Distributed Systems*, 5(1):39–52, January 1994.

[31] S. Midkiff, D. Padua, and R. Cytron. Compiling Programs with User Parallelism. In *Proc. 2nd Workshop on Languages and Compilers for Parallel Computing*, August 1989.

[32] V. S. Pai, P. Ranganathan, S. V. Adve, and T. Harton. An Evaluation of Memory Consistency Models for Shared-Memory Systems with ILP Processors. In *Proc. of the 7th Intl. Conf. on Architectural Support for Programming Languages and Operating Systems*, pages 12–23, 1996.

[33] P. Ranganathan, V. S. Pai, and S. V. Adve. Using Speculative Retirement and Larger Instruction Windows to Narrow the Performance Gap between Memory Consistency Models. In *Proceedings of the 9th Annual ACM Symposium on Parallel Algorithms and Architectures*, pages 199–210, 1997.

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

[35] Sparc International. *The SPARC Architecture Manual*, 1993. Version 9.

[36] Sun Microsystems Inc. *The SPARC Architecture Manual*, January 1991. No. 800-199-12, Version 8.