To appear in the Journal of Parallel and Distributed Computing, 1992

Programming for Different Memory Consistency Models*

Kourosh Gharachorloo†, Sarita V. Adve‡,
Anoop Gupta†, John L. Hennessy†, and Mark D. Hill‡

†Computer System Laboratory
Stanford University
Stanford, California 94305

‡Computer Sciences Department
University of Wisconsin
Madison, Wisconsin 53706

ABSTRACT

The memory consistency model, or memory model, supported by a shared-memory multiprocessor directly affects its performance. The most commonly assumed memory model is sequential consistency (SC). While SC provides a simple model for the programmer, it imposes rigid constraints on the ordering of memory accesses and restricts the use of common hardware and compiler optimizations. To remedy the shortcomings of SC, several relaxed memory models have been proposed in the literature. These include processor consistency (PC), weak ordering (WO), release consistency (RCsc/RCpc), total store ordering (TSO), and partial store ordering (PSO). While the relaxed models provide the potential for higher performance, they present a more complex model for programmers when compared to SC.

Our previous research has addressed this tradeoff by taking a programmer-centric approach. We have proposed memory models (DRF0, DRF1, PL) that allow the programmer to reason with SC, but require certain information about the memory accesses. This information is used by the system to relax the ordering among memory accesses while still maintaining SC for the programmer. Our previous models formalized the information that allowed optimizations associated with WO and RCsc to be used. This paper extends the above approach by defining a new model, PLpc, that allows optimizations of the TSO, PSO, PC, and RCpc models as well. Thus, PLpc provides a unified programming model that maintains the ease of reasoning with SC while providing for efficiency and portability across a wide range of proposed system designs.

1. Introduction

A *memory consistency model* or a *memory model* for a shared-memory multiprocessor system is a formal specification of how memory read and write accesses of a program will appear to execute to the programmer [2, 5]. *Sequential consistency* (SC) [11] is the most commonly used memory model since it requires the execution of a parallel program to appear as some interleaving of the execution of the parallel processes on a sequential machine. While SC allows for simple reasoning about programs, it restricts many common uniprocessor hardware and compiler optimizations that reorder or overlap the execution of memory accesses [4, 13].

* The Stanford University authors are supported by DARPA contract N00039-91-C-0138. Kourosh Gharachorloo is partly supported by Texas Instruments. Anoop Gupta is partly supported by a NSF Presidential Young Investigator Award with matching funds from Sumitomo, Tandem, and TRW.
The University of Wisconsin authors are supported in part by a National Science Foundation Presidential Young Investigator Award (MIPS-8957278) with matching funds from A.T. & T. Bell Laboratories, Cray Research Foundation and Digital Equipment Corporation. Sarita Adve is also supported by an IBM graduate fellowship.

To achieve better system performance, researchers have proposed alternate memory models: *processor consistency* (PC) [5][1], *total store ordering* (TSO) [15], *partial store ordering* (PSO) [15], *weak ordering* (WO) [4], and *release consistency* (RCsc/RCpc) [5]. The optimizations allowed by these models can provide substantial improvement in system performance, especially as memory latencies grow relative to processor speeds [6, 7]. However, the formal definitions of the models are presented almost completely in terms of the optimizations allowed. Defining models in this manner leads to two problems: (1) every new hardware optimization potentially results in a new model, requiring programmers to reason with a wide variety of specifications, and (2) the models are too *hardware-centric* since they mainly express the hardware optimizations they allow; most programmers do not want to deal directly with hardware optimizations.

Our previous research has addressed the above dilemma by proposing a more *programmer-centric* approach that provides a higher level of abstraction to the programmer. This approach guarantees SC if certain information about the memory accesses is provided; the information is used to exploit various optimizations without violating SC. The data-race-free-0 (DRF0) [1] and data-race-free-1 (DRF1) [2] memory models, and the notion of properly labeled (PL) programs[2] [5] allow the programmer to reason with SC, and at the same time allow the optimizations of WO and RCsc. This is achieved by requiring the programmer to explicitly identify the accesses in the program that could be involved in a race.

Our new memory model, called PLpc[3], extends our previous framework to allow the programmer to reason with SC, while allowing the additional optimizations of TSO, PSO, PC and RCpc as well. It achieves this by requiring the programmer to additionally identify a common class of synchronization accesses where one process waits on a location until a particular value is written by another process.

A direct impact of our work is that programmers who prefer to reason with SC need only provide program-level information (as specified by the PLpc memory model) to exploit the same optimizations as allowed by the hardware-centric models. We claim that providing this type of information is easier and more natural for the programmer than reasoning with the hardware-centric models directly. More broadly, the PLpc memory model unifies a large set of seemingly different systems for both programmers and system designers. For programmers, writing programs for the PLpc memory model allows for simple reasoning and portability across many systems. For system designers, specifying PLpc as the memory model allows building systems with a wide range of optimizations without sacrificing portability or ease of use.

---

1. The processor consistency model considered in this paper is different from that proposed by Goodman [9].
2. We will also use PL to imply a memory model that guarantees SC to all PL programs.
3. The PL memory model encompasses systems that guarantee SC among competing accesses (defined later) [5]. The PLpc memory model extends PL to include systems that at most guarantee PC among such accesses. Thus the name PLpc.

The rest of the paper is organized as follows. Section 2 defines the PLpc memory model. Section 3 provides intuition for the optimizations that can be exploited by a system that obeys the PLpc memory model. Section 4 gives mappings from PLpc to the hardware-centric models, which allow programs written for PLpc to be run efficiently on the hardware-centric models without violating SC. Section 5 concludes the paper.

The proofs to support the material in Sections 3 and 4 appear in another paper [3]. The paper develops a formal and general framework for defining, implementing, and proving equivalences among several memory models [3]. It illustrates the use of this framework by deriving a set of sufficient conditions for systems that satisfy the PLpc memory model and proving that the hardware-centric models satisfy these conditions with the mappings of Section 4.

2. The PLpc Memory Model

This section presents the PLpc memory model. Section 2.1 gives a categorization of shared memory accesses that forms the basis of the information required by the PLpc memory model. Section 2.2 states how the information on the category of an access can be correctly conveyed for PLpc and formalizes the PLpc memory model.

2.1. Categorization of Memory Accesses

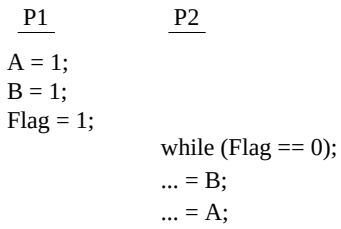
The PLpc memory model uses two notions to categorize memory accesses. First, it distinguishes between accesses that may be involved in a race (usually synchronization accesses) and those that are never involved in a race (usually data accesses). Second, of the accesses that may be involved in a race, PLpc identifies a class of synchronization accesses where a processor repeatedly reads a location until another processor writes a particular value to that location. In this section, we formalize the above access categories. Section 3 discusses the optimizations that such a categorization makes possible.

We start by clarifying some terminology that will be used in the rest of this section. A memory access, or simply an access, is a single read or write to a specific shared memory location.[4] For every execution of a program, the program text defines a partial order, called the *program order* ($\xrightarrow{po}$), on the memory accesses of each process in the execution [17]. An *SC execution* refers to an execution of a program on an SC system (Refer to [2, 3] for formal definitions of a program, an execution, and a system.) A system is SC if the result of every execu-

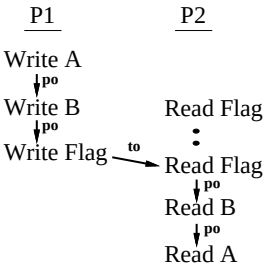
4. An atomic read-modify-write is treated as a read access followed by a write access [2, 5]. We implicitly assume an implementation that does not allow a write to be executed between the read and the write of a read-modify-write to the same location [3].

tion on it can be obtained by some total order ($\xrightarrow{to}$) of the memory accesses of the execution such that $\xrightarrow{to}$ obeys $\xrightarrow{po}$ [11]. The result of an execution has been interpreted as the values its reads return [2, 8]. Finally, two accesses are considered *conflicting* if they are to the same location and at least one of them is a write [17].

Figure 1(a) shows a typical producer/consumer interaction that we use as an example to illustrate the accesses categories. The producer (P1) writes two locations and sets a flag, while the consumer (P2) waits for the flag to be set and then reads the two locations. We next formalize the access categories.



(a) Program Text



(b) Program Order and Total Order

Figure 1. Example code segment.

In Figure 1(a), the accesses to A (and B) are always separated or ordered by the accesses to flag. We refer to the accesses to A and B as *non-competing accesses* and to those to flag as *competing accesses*. The formal definitions follow.

Definition 1: Ordering Chain: For two conflicting accesses u and v of an SC execution with a total order $\xrightarrow{to}$, an ordering chain exists from access u to access v if $u \xrightarrow{po} v$ or

$$u \xrightarrow{po} w_1 \xrightarrow{to} r_1 \xrightarrow{po} w_2 \xrightarrow{to} r_2 \xrightarrow{po} w_3 \cdots \xrightarrow{to} r_n \xrightarrow{po} v,$$

where $n \geq 1$, w_i is a write access, r_j is a read access, and w_i and r_j are to the same location if $i = j$. If all accesses in the above chain are to the same location, then u maybe the same as w_1, and v maybe the same as r_n as long as there is at least one $\xrightarrow{po}$ arc in the chain.[5]

5. Similar ordering relations are also used in [1, 2, 8].

Definition 2: Competing and Non-competing Accesses: Two conflicting accesses of an execution of a program form a *competing pair* if there is at least one SC execution of the program where there is no ordering chain between the accesses. An access is a *competing access* if it belongs to at least one competing pair. Otherwise, it is *non-competing*.[6]

Figure 1(b) shows the $\xrightarrow{po}$ and $\xrightarrow{to}$ arcs that provide an ordering chain between the accesses to A and to B in every SC execution of the code in Figure 1(a). There is no ordering chain between the write and the reads to flag. Therefore, the accesses to A and B are non-competing, while those to flag are competing.

In Figure 1(a), P2 reads flag in a loop until P1 writes it. Such a loop is called a synchronization loop construct. Below, we formalize a simple case of a synchronization loop construct in which the loop repeatedly executes a read or a read-modify-write to a specific location until it returns one of certain specific values. In the appendix, we give a more general definition that, for example, allows implementations of locks using test&test&set [14] or back-off [12] techniques to be considered as synchronization loop constructs.

Definition 3: Synchronization Loop Construct: A *synchronization loop construct* is a sequence of instructions in a program that satisfies the following. (i) The construct executes a read or a read-modify-write to a specific location. Depending on whether the value returned is one of certain specified values, the construct either terminates or repeats the above. (ii) If the construct executes a read-modify-write, then the writes of all but the last read-modify-write store values returned by the corresponding reads. (iii) The construct terminates in every SC execution.

Given that a synchronization loop construct eventually terminates, the number of times the loop executes or the values returned by its unsuccessful reads cannot be detected and should not matter to the programmer. For example, in Figure 1(a), it cannot be detected and does not matter how many times P2 reads flag unsuccessfully, or even what values the unsuccessful reads return, as long as eventually a read returns 1 and terminates the loop. For this reason, we assume that the unsuccessful reads of a synchronization loop construct do not contribute to the result of an SC execution. Thus, if all the accesses of a synchronization loop construct are replaced with only the last read or read-modify-write that exited the loop, we still get an SC execution with the same result as before [3]. Therefore, in analyzing SC executions, we treat a synchronization loop construct as a single access which is the last read or read-modify-write that terminates the loop construct.

Synchronization loop constructs often have another special property. Generally, accesses in a competing pair can execute in any order. However, with the competing pair comprising of the write that terminates a synchronization loop construct and the final read of the loop construct, the write always executes before the read. For example, in Figure 1(a), P1's write to flag must execute before P2's final read of flag. Such write-read competing pairs for which the order of execution is fixed allow for optimizations that are not possible for other accesses. We call such writes and reads loop accesses and formalize them below.

---

6. DRF1 and PL defined the notion of data races [1, 2] and competing accesses [5] that are similar to competing accesses defined above. The major difference is that Definition 1 allows $u = w_1$ and $v = r_n$ only if all accesses on the ordering chain are to the same location. DRF1 and PL do not have such a restriction.

Definition 4: Loop and Non-Loop Reads: A competing read is a *loop read* if (i) it is the final read of a synchronization loop construct that terminates the construct, (ii) it competes with at most one write in any SC execution, and (iii) if it competes with a write in an SC execution, then it returns the value of that write; i.e., the write is necessary to make the synchronization loop construct terminate. A competing read that is not a loop read is a *non-loop read*.

Definition 5: Loop and Non-Loop Writes: A competing write is a *loop write* if it competes only with loop reads in every SC execution. A competing write that is not a loop write is a *non-loop write*.

In summary, we categorize accesses as competing or non-competing. We further categorize the competing accesses as loop or non-loop. Figure 2 summarizes this categorization. Although our definitions may seem complex, the intuition is fairly simple and the information we require, i.e., whether an access is competing or not and whether it is a loop access or not, is naturally available to the programmer in most cases.

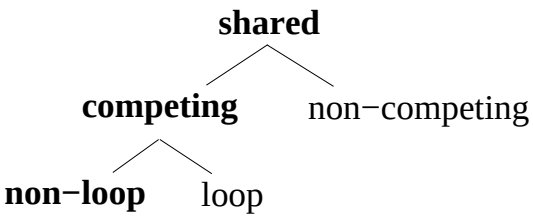


Figure 2. Categorization for read and write accesses to shared data.

2.2. Properly-Labeled Programs

We discuss how access categories can be correctly conveyed for the PLpc memory model by formalizing the notion of a PLpc program, and then define the PLpc memory model.

There are no restrictions on how the category of an access can be conveyed. Several methods are discussed in our earlier work which apply to PLpc as well [1, 2, 5]. One such method is for a system to provide easy to use high-level synchronization constructs (e.g., monitors and distributed task queues), and to require programmers to order all conflicting memory accesses through these constructs. With such an approach, only the writers of the high-level synchronization constructs need to see the full complexity of PLpc. Irrespective of how the categories are actually conveyed, we can use an abstraction where every access has a *label* associated with it that identifies its category. The valid labels for PLpc are non-competing, loop and non-loop. Note that labeling does not require adding extra instructions in the program; it only requires distinguishing memory accesses according to their categories.

We now discuss when accesses are considered correctly labeled. If every access is labeled with the category it actually belongs to, the labeling is clearly correct. However, such information may not be available for certain accesses. For those cases, we allow the labeling to be conservative. This ensures correctness at the cost of not fully exploiting the potential for performance. Informally, a label is conservative if its category requires less information than the actual category of the access. Conservative labels are shown in bold in Figure 2. Thus, a

non-competing access can be conservatively labeled as competing (loop or non-loop) and a loop access can be conservatively labeled as non-loop. The following formalizes when a program is properly labeled. We use the subscript L to distinguish the label for an access from the intrinsic category of the access.

> **Definition 6: Properly-Labeled Programs (PLpc programs):** A program is *properly-labeled (PLpc)* if (i) all accesses labeled $non-competing_L$ are non-competing accesses, and (ii) all accesses labeled $loop_L$ are loop accesses.[7]

To illustrate proper labeling, Figure 3 shows two common synchronization scenarios (shared locations are shown starting with an upper case letter). Figure 3(a) shows the implementation of locks and unlocks using test&set and unset. The while loop containing the test&set forms a synchronization loop construct; therefore, we ignore unsuccessful test&sets. The test of a final test&set competes only with the unset that is required for the loop to terminate. Therefore, the test is a loop read. The set of a final test&set is non-competing. The unset competes only with a final test; therefore, it is a loop write.[8]

```
while ( test&set(S) );          test&set(S)   loop read &
                                              non−competing write
accesses to data
unset(S);                       write(S)      loop write
```

(a) critical section

```
accesses to data
/* barrier code */
local_flag = ! local_flag;
if ( fetch&incr(Count) == N ) {     fetch&incr(Count)   non−loop read &
                                                        non−loop write
        Count = 1;                  write(Count)        non−competing write
        Flag = local_flag;          write(Flag)         loop write
} else {
        while ( Flag != local_flag ) ;   read(Flag)     loop read
}
accesses to data
```

(b) barrier

Figure 3. Examples of access categories and access labels.

Figure 3(b) shows the implementation of a barrier [12] using an atomic fetch-and-increment operation [10]. The while loop containing the reads on flag forms a synchronization loop construct; therefore, we ignore the unsuccessful reads of flag. The write to count is a non-competing access; the fetch and increment on count, and the write and the final read to flag are competing accesses. The read and write to flag are loop accesses, while the

7. Condition (i) is very similar to that for PL [5] and DRF1 [2].

8. DRF1 and PL did not ignore unsuccessful accesses of a synchronization loop construct. An unsuccessful set can compete with an unset and was therefore considered competing. To differentiate between an unset and set, competing accesses were further categorized as paired and unpaired synchronization in DRF1, and syncs and nsyncs in PL. PLpc does not need these distinctions.

fetch and increment are non-loop accesses.

The PLpc model is defined as follows.

Definition 7: The PLpc Memory Model: A system obeys the PLpc memory model if all executions of any PLpc program on the system are SC executions.

3. Possible Optimizations with the PLpc Memory Model

This section provides the intuition for how the information in a PLpc program can be used by a system obeying the PLpc memory model to exploit certain optimizations both in the hardware and compiler while still maintaining SC. A more general set of optimizations allowed by the PLpc memory model along with a proof of correctness appears in [3]. Below we first state a set of sufficient requirements for SC and then show how these requirements can be weakened for systems that satisfy only the PLpc memory model.

To provide SC executions, it is sufficient if (i) accesses of a single processor are executed one at a time in program order, and (ii) a write is made visible to all processors simultaneously (referred to as *atomicity*) [16]. If (i) is satisfied, it follows that a weaker notion of the atomicity condition in (ii) suffices: it is sufficient if a write becomes visible simultaneously to all processors other than the one that issued the write. In the following, we use *atomic* to refer to this weaker notion. The information in a PLpc program allows a system obeying the PLpc memory model to reorder and overlap certain accesses of a processor and to allow certain writes to be non-atomic.

The first level of information provided in PLpc programs distinguishes between competing and non-competing accesses. This allows for virtually the same optimizations as the PL and DRF1 models. PL and DRF1 allow for all optimizations of WO and RCsc [2, 5]. Thus, non-competing accesses can be fully reordered and overlapped (as long as intra-processor dependences are observed) in the region between a competing read and a competing write.[9] However, with just this level of information, competing accesses have to be executed as on an SC system (as is the case for WO and RCsc).

The second level of information provided in PLpc programs decreases the restrictions on competing accesses. This results in potentially higher performance than WO and RCsc, and allows the optimizations of TSO, PSO, PC and RCpc as discussed in Section 4. Specifically, we have shown that a loop read does not have to wait for previous writes, and a non-loop read does not have to wait for previous loop writes [3]. Equivalently, when a read follows a write, the read need wait for the write only if both are non-loop accesses. Further, we have also shown that loop writes do not have to be atomic [3]. While we do not give the proof for these optimizations here,

9. In the discussion, relations among accesses such as between, previous, following, etc., are in the context of program order.

the following examples convey the intuition for why they work, and also indicate the performance gains they yield.

Consider the code segment shown in Figure 4(a). Each processor produces a value, sets a flag, waits for the other processor's flag to be set, and consumes the value produced by the other processor. The read and write accesses to the flag locations are competing loop accesses and are shown in bold. As long as we ignore the unsuccessful intermediate reads in the loops, the executions of the program appear SC even if loop reads overlap previous loop writes. Consider the performance gain from this optimization. Assume P1 has set Flag1 first. Now assume P2 sets Flag2. The optimization allows P2 to continue with the loop-read before the write of Flag2 completes, in effect overlapping and hiding the latency of the write with the following computation.[10]

P1	P2
A = 1;	B = 1;
Flag1 = 1;	**Flag2** = 1;
while (**Flag2** == 0);	while (**Flag1** == 0);
... = B;	... = A;

(a)

P1	P2
A = 1;	B = 1;
unlock L1;	**unlock L2;**
lock L2;	**lock L1;**
... = B;	... = A;

(b)

Figure 4. Example code segments with loop reads and loop writes.

As another example, consider the code segment shown in Figure 4(b). The code is similar to Figure 4(a) except that it uses locks and unlocks instead of flags. Assume the implementation of locks and unlocks shown in Figure 3(a). PLpc allows the read (loop read) and the write (non-competing write) of the test-and-set to occur before the previous unset (loop write). Thus, the acquiring of one lock can occur fully before the previous release of another lock without violating SC and provides performance gains similar to those described for the previous example.

10. Either P2 has a valid copy of Flag1 and successfully completes the loop and continues or it needs to fetch a copy of Flag1 and thus overlaps the latency of this read with the previous write.

Finally, the loop writes to the flags in Figure 4(a), to the lock locations in Figure 4(b), or to the flag in the barrier code in Figure 3(b) need not be atomic to get SC executions. The writes in these examples would benefit from using updates (versus invalidates) in a cache-coherent environment to reduce the communication latency for the critical synchronization. However, supporting atomic updates in a large-scale system is both unnatural and inefficient [5]. Thus, there is clear benefit from allowing certain competing writes to be non-atomic with the guarantee of SC results.

In summary, PLpc programs provide two types of information about shared accesses. The information distinguishing between competing and non-competing accesses allows for virtually the same optimizations as PL and DRF1. The second type of information (i.e., loop and non-loop category) allows for relaxing the ordering between competing accesses further while still maintaining SC. The gain from the latter optimization becomes more pronounced in programs with a high frequency of competing accesses (i.e., with fine grain synchronization).

Although we have mainly concerned ourselves with hardware optimizations in the above examples, optimizations possible in the compiler are directly related to the flexibility the hardware has in ordering memory accesses. Due to its extra flexibility, PLpc has the potential of allowing further compiler optimizations than allowed by PL and DRF models as well. However, since many compiler optimizations require the full flexibility of reordering both read and write accesses, we believe the extra gain from the limited reordering between competing accesses as allowed by PLpc will be more prominent in hardware than in the compiler.

4. Porting PLpc Programs to Hardware-Centric Models

This section discusses how the information contained in a PLpc program (in the form of access labels) can be used to efficiently execute the program on a system supporting a hardware-centric model while still maintaining SC. This is done by providing a mapping from the access categories supported by PLpc to the access categories supported by the hardware-centric models. This mapping is mechanical, thereby allowing for automatic and efficient portability of PLpc programs to a variety of system architectures.

4.1. Porting PLpc Programs to WO and RCsc

DRF1 and PL already indicate how PLpc programs can be mapped to WO and RCsc [2, 5]. For WO, competing accesses should be mapped to synchronization accesses. For RCsc, competing reads and writes should be mapped to acquires and releases respectively. Since WO and RCsc require acquires and releases to be SC, they do not benefit from the loop versus non-loop categorization of PLpc.

4.2. Porting PLpc Programs to TSO, PSO, PC, and RCpc

TSO, PSO, PC, and RCpc allow reads to bypass previous writes, while PC and RCpc allow writes to be non-atomic. As discussed in Section 3, allowing these optimizations for loop accesses does not violate SC. However, allowing non-loop reads to bypass previous non-loop writes, and making non-loop writes non-atomic can violate SC as illustrated by the examples in Figure 5. In the figure, all accesses are non-loop accesses. For the code segment in Figure 5(a), TSO, PSO, PC, and RCpc allow the reads to execute before previous writes. Thus, the non-SC outcome (x,y) = (0,0) is possible. Similarly, for the code segment in Figure 5(b), non-atomicity of writes makes the non-SC outcome of (u,v,w) = (1,1,0) possible under PC and RCpc.

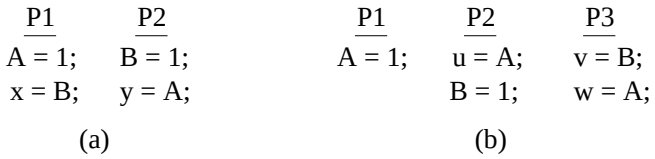


Figure 5. Example code segments to illustrate violation of SC.

TSO, PSO, PC, and RCpc do not provide a direct way in which non-loop reads can be made to stall before previous non-loop writes complete. Furthermore, PC and RCpc do not provide a direct way in which non-loop writes can be made atomic. However, we have shown that using atomic read-modify-writes for certain non-loop accesses can achieve a similar effect by satisfying a set of more general sufficient conditions for a PLpc system [3]. The exact mapping is described below.

For TSO and PSO, for every non-loop write followed by a non-loop read, at least one of the accesses should be part of an atomic read-modify-write operation. Additionally, PSO also allows writes to be reordered if they are not separated by a STBAR (store barrier). Therefore, competing writes (whether looping or non-looping) need to be preceded by a STBAR for PSO. For PC and RCpc, all non-loop reads should be part of an atomic read-modify-write operation. Additionally, for RCpc, competing reads and writes should be identified as acquires and releases respectively.

The above mappings use atomic read-modify-writes for non-loop accesses since the base systems (TSO, PSO, PC, and RCpc) do not provide a more direct way of achieving the orders described in Section 3. It may at first seem unnatural and inefficient to use read-modify-writes instead of regular accesses. However, many synchronization constructs such as locks and barriers are already implemented with atomic read-modify-write operations. In programs that use this type of synchronization, the non-loop accesses are often confined to these operations. Therefore, many programs naturally (and hence efficiently) obey our mapping for non-loop accesses. For example, in Figure 3(b), the competing non-loop accesses to location Count are already part of a fetch-and-increment operation. This explains why many programs written for SC systems work correctly on TSO and PC

systems without any changes.

In the uncommon cases where a competing non-loop access is not naturally part of a read-modify-write, the access may need to be converted into a dummy read-modify-write operation to provide the correct mapping. Converting the normal access into a more expensive read-modify-write access may be inefficient. In some cases, conversion to a read-modify-write may be impossible if the hardware does not provide sufficiently general read-modify-write operations required for the conversion (e.g., a test&set can not be used to provide the functionality of an arbitrary write operation). To solve this problem, a base system (TSO, PSO, PC, or RCpc) can be extended to provide direct mechanisms for achieving the required order among competing accesses. For TSO and PSO, supporting a fence mechanism [5] to delay future reads for previous writes allows for directly delaying non-loop reads for previous non-loop writes. For PC and RCpc, the extra ability to force certain writes to appear atomic provides an alternative correct mapping: a fence is required between every non-loop write and non-loop read and all non-loop writes need to be mapped to atomic writes.

Of all the hardware-centric models considered in this section, RCpc provides the most aggressive system for translating the information provided in a PLpc program into performance gains while still maintaining SC. However, it is possible to build a more aggressive implementation that obeys the PLpc model based on the sufficient system constraints specified in [3].

5. Conclusions

Sequential consistency (SC) is a simple model for programmers, but restricts the use of common uniprocessor hardware and compiler optimizations. To achieve higher performance, several alternate memory models have been proposed. Unfortunately, this divergence from SC, along with the hardware-oriented description of the models, results in more complicated models for the programmer.

We proposed the memory model PLpc using a more programmer-centric approach to address the tradeoff between simplicity and efficiency. PLpc allows programmers to reason with SC, but requires programmers to explicitly identify accesses that race, and accesses that are part of a common waiting-synchronization construct. These two pieces of information are used to exploit the optimizations proposed by previous models while still maintaining SC. Using PLpc is simple because the required information about the program is often naturally known and conveyed by the programmer.

PLpc serves as a unifying memory model for the various memory access optimizations that have been proposed by hardware and compiler designers. For programmers, PLpc programs can be easily and efficiently ported from one system to another. With the earlier models, this would require programmers to re-establish the correct-

ness of their programs using the hardware specifications of each system. Correspondingly, for system designers, the PLpc model allows for systems with various degrees of optimizations without sacrificing programming simplicity or portability.

Although PLpc allows optimizations proposed by all current hardware-centric models, it is possible to allow more optimizations if more information is known about the program. In [3], we develop a general framework that gives intuition for the type of information that can be used for different optimizations, and also provide specific examples of such information.

Acknowledgements

We thank Phillip Gibbons, Michael Merritt, and the anonymous referees for their comments.

References

1. S. V. Adve and M. D. Hill, Weak Ordering - A New Definition, *Proc. 17th Annual Intl. Symp. on Computer Architecture*, May 1990, 2-14.
2. S. V. Adve and M. D. Hill, A Unified Formalization of Four Shared-Memory Models, Computer Sciences Technical Report #1051, University of Wisconsin, Madison, September 1991. Submitted for publication.
3. S. V. Adve, K. Gharachorloo, M. D. Hill, A. Gupta and J. L. Hennessy, A Framework for Defining, Implementing, and Proving Equivalences Among Memory Models, Computer Sciences Technical Report, University of Wisconsin, Madison, To be published.
4. M. Dubois, C. Scheurich and F. A. Briggs, Memory Access Buffering in Multiprocessors, *Proc. 13th Annual Intl. Symp. on Computer Architecture 14*, 2 (June 1986), 434-442.
5. K. Gharachorloo, D. Lenoski, J. Laudon, P. Gibbons, A. Gupta and J. Hennessy, Memory Consistency and Event Ordering in Scalable Shared-Memory Multiprocessors, *Proc. 17th Annual Intl. Symp. on Computer Architecture*, May 1990, 15-26.
6. K. Gharachorloo, A. Gupta and J. Hennessy, Performance Evaluation of Memory Consistency Models for Shared-Memory Multiprocessors, *Proc. 4th Intl. Conf. on Architectural Support for Programming Languages and Operating Systems*, 1991, 245-257.
7. K. Gharachorloo, A. Gupta and J. Hennessy, Hiding Memory Latency using Dynamic Scheduling in Shared-Memory Multiprocessors, *Proc. 19th Intl. Symp. on Computer Architecture (to appear)*, 1992.

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

9. J. R. Goodman, Cache Consistency and Sequential Consistency, Computer Sciences Technical Report #1006, Univ. of Wisconsin, Madison, February 1991.

10. A. Gottlieb, R. Grishman, C. P. Kruskal, K. P. McAuliffe, L. Rudolph and M. Snir, The NYU Ultracomputer - Designing an MIMD Shared Memory Parallel Computer, *IEEE Trans. on Computers*, February 1983, 175-189.

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

12. J. M. Mellor-Crummey and M. L. Scott, Algorithms for Scalable Synchronization on Shared-Memory Multiprocessors, *ACM Transactions on Computer Systems*, February 1991, 21-65.

13. S. Midkiff, D. Padua and R. Cytron, Compiling Programs with User Parallelism, *Proceedings of the Second Workshop on Languages and Compilers for Parallel Computing*, August 1989.

14. L. Rudolph and Z. Segall, Dynamic Decentralized Cache Schemes for MIMD Parallel Processors, *Proc. Eleventh Intl. Symp. on Computer Architecture*, June 1984, 340-347.

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

16. C. Scheurich and M. Dubois, Correct Memory Operation of Cache-Based Multiprocessors, *Proc. Fourteenth Annual Intl. Symp. on Computer Architecture*, Pittsburgh, PA, June 1987, 234-243.

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

Appendix

Below, we give a more general definition for synchronization loop constructs than Definition 3.

Definition A: Loop construct: A *loop construct* is a sequence of instructions in a program that would be repeatedly executed until a specific read in the sequence (the *exit read*) reads a specific location (the *exit location*) and returns one of certain values (the *exit read values*). If the exit read is part of a read-modify-write, then the write of the read-modify-write is called the *exit write* and the value it writes is called the *exit write value*.

Definition B: Synchronization Loop Construct: A loop construct in a program is a synchronization loop construct if it always terminates for every SC execution of the program, and if the following holds. Consider a modification of the program so that it executes beginning at the loop construct. Add another process to the program that randomly changes the data memory. Consider every SC execution with every possible initial state of the data memory and processor registers. At the beginning of every such SC execution, the exit read, exit location, and exit read values should only be

a function of the initial state of memory and registers, and of the program text. The exit write value can additionally be a function of the value that the exit read returns. Then for every such SC execution,

(i) except for the final exit write, loop instructions should not change the value of any shared memory location,

(ii) the values of registers or private memory changed by any loop instruction cannot be accessed by any instruction not in the loop construct,

(iii) a loop instruction cannot modify the exit read, exit location, exit read values, or the exit write values corresponding to a particular exit read value,

(iv) the loop terminates only when the exit read returns one of the exit read values from the exit location and the exit write stores the exit write value corresponding to the exit read value returned,

(v) if exit read values persist in the exit location, then the loop eventually exits.

When analyzing an SC execution for access categories, all accesses of a synchronization loop construct can be replaced only by the final successful exit read and exit write [3]. The unsuccessful accesses may be labeled non-competing since non-competing accesses allow the most aggressive optimizations.

The above definition is fairly general (which is a reason for its complexity). For example, it allows implementations of locks using test&test&set [14] or back-off [12] techniques to be considered as synchronization loop constructs.