McGill University
School of Computer Science
ACAPS Laboratory

Advanced Compilers, Architectures and Parallel Systems

Location Consistency: Stepping Beyond the Barriers of Memory Coherence and Serializability

Guang R. Gao
Vivek Sarkar †

ACAPS Technical Memo 78
December 31, 1993
Revised December 31, 1994

A condensed version of this paper appears in the proceedings of the 1995 International Conference on Parallel Processing.

gao@cs.mcgill.ca, vivek_sarkar@vnet.ibm.com

†IBM Santa Teresa Laboratory, 555 Bailey Avenue, San Jose, California 95141

Abstract

A memory consistency model represents a binding "contract" between software and hardware in a shared-memory multiprocessor system. It is important to provide a memory consistency model that is easy to understand and that also facilitates efficient implementation.

The memory consistency model that has been most commonly used in past work is *sequential consistency* (SC), which requires the execution of a parallel program to appear as some interleaving of the memory operations on a sequential machine. To reduce the rigid constraints of the SC model, several relaxed consistency models have been proposed, notably *weak ordering* (or *weak consistency*) (WC), *release consistency* (RC), *data-race-free-0*, and *data-race-free-1*. These models allow performance optimizations to be correctly applied, while guaranteeing that sequential consistency is retained for a specified class of programs. We call these models *SC-derived* models. A central assumption in the definitions of all SC-derived memory consistency models is the *memory coherence* assumption, which can be informally stated as follows [13]: "all writes to the same location are serialized in some order and are performed in that order with respect to any processor".

In this paper, we propose a *location consistency* (LC) model for memory consistency. A distinguishing feature of the LC model is that it does not rely on the assumption of memory coherence. Instead of assuming that all writes to the same location are serialized according to some total order, we model the state of a memory location as a partially ordered multiset (pomset) of write operations and synchronization operations. The partial order in the pomset naturally follows from the ordering constraints defined by a concurrent program and its execution. The LC model provides a very simple "contract" between software and hardware; only the partial order defined by the program needs to be obeyed, and there is no requirement for all processors to observe the same ordering of concurrent write operations. Further, concurrent asynchronous reads and writes to the same memory location are well-defined in the LC model, rather than being considered as error conditions. We demonstrate that the LC model allows more compiler and hardware performance optimizations to be applied, compared to the other memory consistency models mentioned above. The LC model is also easier to implement because the serialization constraints imposed by SC and SC-derived models (e.g. due to the memory coherence assumption) are removed.

Contents

List of Figures

1 Introduction

A memory consistency model represents a binding "contract" between software and hardware in a shared-memory multiprocessor system. It provides a formal specification of how the memory system can behave in response to concurrent operations, and thus defines the permitted orderings of events across different processors. These events include memory read and write (load and store) operations and synchronization operations. Apart from providing a behavioral specification, the abstractions used to define a memory consistency model can have a big impact on the ease of developing and reasoning about concurrent programs, as well as on the ease of developing correct and efficient implementations of the model. It is important for the model to provide a simple working definition of memory consistency that is easy to understand and also easy to implement efficiently.

There is a noticeable dichotomy between recent software and hardware viewpoints of memory consistency in shared-memory multiprocessors. The memory consistency model typically assumed by software is a dependence-based model, as exemplified by program dependence graphs [11] and parallel program graphs [23] and various high-level programming languages that can be translated to these program graph representations. In the dependence-based model, memory operations are partially ordered, with the constraint that there be no concurrency among conflicting accesses directed to the same memory location. As discussed below, the memory consistency models typically assumed by hardware are based on the notions of memory coherence and serializability [19, 13, 2], which assume that (possibly concurrent) memory operations on the same location are totally ordered at run-time and that the same total ordering is observed by all processors. Though it is essential to know exactly what assumptions are made by the hardware regarding memory coherence to understand how a given concurrent program will behave, these issues have largely been ignored in the software viewpoint. In fact, as discussed in Section 2, compiler optimization algorithms such as instruction scheduling and register allocation will not work correctly for most memory consistency models assumed by hardware. The goal of this paper is to bridge the gap between the software and hardware viewpoints of memory consistency. In this paper, we propose a *location consistency* (LC) model for memory consistency which combines ingredients from the "partial order" program models in the software viewpoint with the "concurrent access" models in the hardware viewpoint. We believe that the LC model provides an efficient and usable memory consistency model that reconciles the differences between the hardware and software viewpoints, and that can indeed serve as a workable binding contract between software and hardware.

The hardware memory consistency model that has been most commonly used as a basis for past work is *sequential consistency* (SC) [19]. It has been argued that the SC model retains the simplicity of the uniprocessor model because it is a direct extension of the uniprocessor memory abstraction to multiprocessor systems. As observed in [2], a sequentially consistent multiprocessor behaves like a multiprogrammed uniprocessor. However, it has been found that sequential consistency limits performance by preventing the use of common uniprocessor hardware optimizations such as store buffers and out-of-order memory operations [13, 1]. In addition, we observe that sequential consistency prevents the use of common uniprocessor compiler optimizations such as instruction scheduling and register allocation (this observation is discussed in more detail in Section 2).

The main approach taken in recent work on memory consistency models is to define weaker models that allow performance optimizations to be correctly applied, while also guaranteeing that the weaker model is sequentially consistent for a specified class of programs. Therefore, we will refer to these weaker models as *SC-derived* models. Recently proposed SC-derived memory consistency models include *weak ordering* (WC) [10] , *release consistency* (RC) [13], *data-race-free-0* [1], and *data-race-free-1* [2]. These models allow performance optimizations to be correctly applied, while guaranteeing that sequential consistency is retained for a specified class of programs. In this sense, it has been observed in [2] that the SC-derived models are very similar. A more detailed discussion of these SC-derived models is provided in Section 7.

A central assumption in the definitions of all SC-derived memory consistency models is the *memory coherence* assumption, which can be stated as follows [13]: "all writes to the same location are serialized in some order and are performed in that order with respect to any processor". Memory coherence is a less restrictive form of serializability — it enforces serializability of operations performed on the same memory location, rather than serializability of all memory operations.

The central assumptions of our work are very different from that of the previous approaches outlined above. A distinguishing feature of the LC model is that it does not rely on the assumption of memory coherence. Instead of assuming that all writes to the same location are serialized according to some total order, we model the state of a memory location as a partially ordered multiset (pomset) [22] of write operations and synchronization operations. Unlike the position taken by the SC-derived models, we do not believe that the sequential consistency model is simple or natural to the programmer's way of thinking when writing concurrent programs, for the following reasons.

First, the definition of sequential consistency is based on total orderings of concurrent events, which are required to be serializable. We believe that serializability is an unnatural assumption to make for concurrent programs, in which events may intrinsically be partially ordered. As mentioned above, the SC-derived models impose a memory coherence assumption which enforces serializability of operations performed on the same memory location, rather than serializability of all memory operations.

Second, sequential consistency is far from simple because it forces the programmer to deal with complex interactions among concurrent regions of a program e.g. consider the subtleties and pitfalls identified in various attempted solutions to the critical section problem, and the complex reasoning about inter-processor interactions required to prove the correctness of Dekker's solution [26]. Apart from making programmer errors more likely, we believe that the inherent complexity of understanding parallel program execution under the SC model naturally makes it more difficult to implement software and hardware optimizations for the SC model.

Third, the definition of memory consistency depends on the notion of a memory operation being "globally performed", for both sequential consistency and recently proposed weaker consistency models [13]. The notion of "globally performed" implies some idealized assumption about global time in order to explain the serialization of events. As explained in section 7, the notion of "globally performed" is hard to interpret and maintain in a multiprocessor system.

In contrast to the SC and SC-derived models, we model the state of a memory location as a partially ordered multiset (pomset) [22]. Each element of the pomset corresponds to either a write operation to the location or a synchronization operation. The partial order in the pomset naturally follows from the ordering constraints defined by a concurrent program and its execution. In our model, a write operation to memory location L with value v from processor p updates the state of L by inserting an element $write(p, v)$ in the pomset. An ordering is created between two write operations in the pomset if they orginate from the same processor. A synchronization operation inserts an element in the pomset of all memory locations affected by the synchronization. An ordering is created between a synchronization operation and a write operation if the processor that originates the write operation also participates in the synchronization. The state of a memory location can be observed by read operations: a read operation of memory location L returns one of a set of possible values from the pomset corresponding to L.

Informally, we say that a multiprocessor system is *location consistent* (LC) if each read operation that is executed on the multiprocessor returns a value that belongs to the value set specified by the state of the memory location as described above. A formal definition of LC is given in section 5, Definition 5.1. Note that the pomset model for the state of a memory location is solely an abstraction used for defining the LC model. It is not intended that the pomset abstraction and the accompanying state-update rules be actually implemented on the multiprocessor system.

The LC model provides a very simple "contract" between the programmer/compiler and the hardware. Memory operations (even to the same location) need not be serialized, only the partial order defined by the program needs to be maintained. The LC model allows more performance optimizations to be applied, compared to the other memory consistency models mentioned above. The LC model is also easier to implement because the constraints imposed by the sequential consistency model are removed. In our model, the observable state of the memory location does not have to appear the same among different processors, thus relaxing the constraints imposed by other memory consistency models. As explained later in section 6, support for the LC model is already available by default in many existing shared-memory multiprocessors. Further, the flexibility offered by the LC model will simplify the implementation of cache coherence support and provide the potential for additional performance improvements. In addition, the LC model is applicable naturally to a more general class of concurrent programs than considered by the other memory consistency models mentioned above e.g. asynchronous concurrent algorithms such as successive-over-relaxation.

One distinguishing feature of our work is the emphasis on both hardware optimizations and compiler optimizations. Many past memory consistency models arose from hardware experience with building shared-memory multiprocessor architectures. Considerations related to the program model and compiler optimization were relatively neglected. For example, the interpretation of "globally performed", essential for understanding previous memory consistency models, is tied closely to understanding the underlying hardware memory architecture [28]. We believe that the recent progress made in compiler optimization techniques for uniprocessor architectures deserve equal attention when defining a memory consistency model. As explained in Section 2, SC and SC-derived models often put unnecessary constraints to the ordering of memory operations. These constraints may become limiting factors when trying to apply compile-time transformations such as instruction scheduling and register allocation.

The rest of this paper is organized as follows. In section 2, we illustrate the restrictions on performance optimizations imposed by the SC and SC-derived models through some motivating examples. In section 3, we summarize the program model assumed in this paper. Section 4 defines the pomset abstraction of memory systems used in our work. Section 5 then defines the location consistency (LC) model, which is the new memory consistency model introduced in this work. The performance potential and implementation implications of the LC model are discussed in section 6. In section 7, we discuss other related memory consistency models and compare them with the LC model. Finally, our conclusions are presented in section 8.

2 Motivating Examples

In this section, we illustrate the restrictions on performance optimizations imposed by the SC and SC-derived models through some motivating examples. We also try and identify why these performance restrictions do not arise in the LC model; however, it may be necessary to read Sections 3, 4, 5 to fully appreciate how these examples would work under the LC model.

Let us illustrate the constraints imposed by sequential consistency by the simple example in Figure 1, which was also presented in [2]. Under the sequential consistency model, the memory operations from processor 1 and 2 appear to be "serialized" in some order which is observed by both processors. Therefore, in this example, it is easy to show that the result $r1 = r2 = 0$ is prohibited by sequential consistency. On the other hand, under the location consistency model defined in this paper, there is no such serialization constraints: the writes to X, Y, $r1$, $r2$ can be observed in different orders by different processors, since they update different locations. Thus, the result $r1 = r2 = 0$ is perfectly legal under the location consistency model.

We now illustrate the constraints imposed by the memory coherence assumption using the program example shown in Figure 2. In Section 1, we observed that the SC and SC-derived models make a strong assumption on memory coherence, which amounts to assuming that all writes to the same memory location are serialized in some order and are performed in that order with respect to all processors [13]. Note that all read and write operations are targeted to the same memory location L. The initial value of L is 0, and the two concurrent writes store the values 1 and 2 into L. Let R be a set of tuples such that $R = \{(0,0)$, (0,1), (0,2), (1,1), (1,2), (2,1) $(2,2)\}$. Under the LC model, since there are no constraints due to the memory

```
X := 0; Y := 0; /* initial values */

Processor 1     Processor 2
-----------     -----------
  X := 1;         Y := 1;
 r1 := Y;        r2 := X;

NOTE: result r1 = r2 = 0 is prohibited by sequential consistency
```

Figure 1: Simple example of constraints imposed by sequential consistency

```
L := 0; /* initial value */

Processor 1     Processor 2     Processor 3     Processor 4
-----------     -----------     -----------     -----------
w1: L := 1;     r1: read L;     w2: L = 2;      r3: read L;
                . . .                           . . .
                r2: read L;                     r4: read L;
```

Figure 2: Simple example of concurrent reads and writes to same memory location

coherence assumption, the permissible values that can be returned by the two read operations in processor 2 are $(r1, r2) \in R$, i.e. any tuple in $R = \{(0,0)$, (0,1), (0,2), (1,1), (1,2), (2,1) $(2,2)\}$. The two read operations in processor 4 are symmetric to but independent of the two read operations in processor 2. Therefore the permissible values of $r1, r2, r3, r4$ under the LC model are given by $(r1, r2, r3, r4) \in R \times R$. In the SC and SC-derived models, the permissible values for $(r1, r2, r3, r4)$ will be a more restricted subset of $R \times R$, due to the ordering constraint imposed by memory coherence. For example, $(r1, r2, r3, r4) = (1, 2, 2, 1)$ cannot be obtained on a sequentially consistent memory model, because the definition of "globally performed" requires that processes 2 and 4 observe the same ordering of write operations $w1$ and $w2$. It is exactly this kind of global ordering constraint that prevents the use of common hardware and compiler optimizations under a sequential consistency memory model.

Figure 3 outlines a slightly more complicated example than Figure 2 by introducing a barrier-style synchronization operation between processors P_1 and P_2, along with concurrent reads and writes to the same memory location. The SC and SC-derived models require that write operations $w1$, $w2$, and $w3$ be observed in the same order by both processors; in particular, the observed order must be one of $< w1, w2, w3 >$, $< w1, w3, w2 >$, $< w3, w1, w2 >$. Assuming that the write operations are completed before any of the read operations, this memory coherence constraint prohibits an execution sequence in which (say) read operation $r1$ returns $val2$, and read operation $r2$ returns $val3$, because both processors are required to observe the same ordering of write operations. However, the scenario that is prohibited is likely to be a

```
 Processor 1          Processor 2
-------------        -------------
w1: L := val1;       w3: L := val3;
    . . .                . . .
w2: L := val2;           . . .
    . . .                . . .
r1: read L;          r2: read L;
    . . .                . . .
    sync(P1,P2) <-----> sync(P1,P2)
    . . .                . . .
    . . .            r3: read L;
```

Figure 3: Example of concurrent reads/writes with synchronization

```
doall j = 1, n
  do i = 1, n
    a(i,j) = (a(i,j) + a(i-1,j) + a(i+1,j) + a(i,j-1) + a(i,j+1))/5
  end do
end do
```

Figure 4: Simple concurrent program

more efficient way of satisfying the memory requests because of the potential for cache locality and/or register locality; it is likely to be more efficient for read operation $r1$ on processor P_1 to pick up the most recent value ($val2$) computed for location L on processor P_1, and for read operation $r2$ on processor P_2 to pick up the most recent value ($val3$) computed for location L on processor P_2. Given the concurrency defined in the program, we see no reason why this scenario should be prohibited by a memory consistency model.

Let us look more closely at the impact of the memory coherence assumption on register allocation, for the example in Figure 3. Register allocation [3] is a well-known compiler optimization in which load and store operations are eliminated by saving values in registers. However, it would be incorrect under the memory coherence assumption to save variable L in a register in processors 1 and 2, because the register allocation prevents one processor from observing changes made to a shared variable by another processor. In contrast, the LC model permits variable L to be saved in a register in both processors 1 and 2 (up to the synchronization point). We will come back to this example in later discussions in Sections 4 and 6.

Figure 4 outlines a simple successive-over-relaxation loop kernel for a 5-point stencil. There are data races among memory operations in this example due to the concurrency among iterations of the `doall j` loop e.g. there is a data race on `a(1,1)` between the write operation on `a(i,j)` in iteration `(j=1,i=1)` and the read operation on `a(i,j-1)` in iteration `(j=2,i=1)`.

These data races are considered to be program errors in compiler-oriented parallel program models such as the program dependence graph [11] and the parallel program graph [23], as well as in systems for detecting access anomalies [9]. The data races are also considered to be program errors in previously defined weak consistency models [2], unless all memory operations on array `a` are classified as synchronizing operations (which naturally increases the overhead of the memory operations). The data races are not considered to be program errors in the sequential consistency model, but that's because the sequential consistency model implicitly treats each memory operation as a synchronizing operation, thus incurring a large overhead for all memory operations.

In contrast, the LC model neither considers the data races to be program errors nor requires the memory operations to be classified as synchronizing operations for correctness. The semantics of data races is well-defined by the partial order semantics of program execution under the LC model. These properties of the LC model are consistent with the assumptions made by designers of asynchronous algorithms [7].

From the previous examples, one can imagine several situations in which the SC model prevents the use of instruction reordering, an optimization that is commonly used in compilers and in uniprocessor hardware. For example, in Figure 1 the result $r1 = r2 = 0$ is prohibited by sequential consistency. An implication of this observation is that, under the sequential consistency model, it is illegal to reorder the store and load operations within a process even though they operate on distinct variables (if reordering is performed in both processes, the result $r1 = r2 = 0$ can be obtained). For SC-derived models, the same restriction holds if the accesses to X and Y are classified as synchronizing operations; but if they are not classified as synchronizing operations, the example program will not belong to the restricted program classes (e.g. properly labeled programs [13]) that are of primary interest to the SC-derived models. Note that, in the LC model, it is perfectly legal to reorder the instructions within a process since they operate on distinct locations.

3 Program Model

In this section, we outline the program model assumed for this work. We do not wish to favor any particular concurrent programming paradigm. The focus of the LC model is on defining memory consistency. To that end, we define a small set of memory and synchronization operations that are relevant to defining memory consistency. Using these memory and synchronization operations as primitives, the LC model can be used to support parallel control structures selected from a variety of different parallel program models such as fork-join [26], cobegin-coend [5], SPMD [6], program dependence graphs [11], and parallel program graphs [23].

The memory and synchronization operations assumed by the LC model that are relevant to defining memory consistency are as follows (the multiprocessor system will have additional operations for intra-processor computations and control flow; but it is assumed that all inter-processor shared-memory and synchronization operations in the multiprocessor system can be defined using some or all of these memory and synchronization operations):

- *Memory write* — if processor P_i wants to write value v in location L, it performs a $write(P_i, v, L)$ operation, which we also represent by the notation, $L := v$, in processor P_i's instruction sequence.

- *Memory read* — if processor P_i wants to read a value from location L, it performs a $read(P_i, L)$ operation, which we also represented by the notation, *read* L, in processor P_i's instruction sequence.

- *Directed control synchronization* — if processor P_2 needs to wait for processor P_1, the synchronization is accomplished by P_1 performing a $signal(P_2)$ operation and by P_2 performing a corresponding $wait(P_1)$ operation. This operation can be efficiently implemented by a binary semaphore [8].

- *Undirected control synchronization* — if processors $P_1, \ldots, P_k$ all need to synchronize among each other, the synchronization is accomplished by each processor performing a $sync(\{P_1, \ldots, P_k\})$ operation. This operation can be efficiently implemented by a counting semaphore [24]. A *sync* operation performed on the entire set of processors is equivalent to a barrier synchronization.

- *Producer-consumer data synchronization* — if processors $P_2, \ldots, P_k$ need to wait for a value produced at location L by processor P_1, the synchronization is accomplished by P_1 performing a $sync\_write(L)$ operation and processors $P_2, \ldots, P_k$ performing a $sync\_read(L)$ operation. As in other work [4, 27], we assume a single-writer multiple-reader model for this form of producer-consumer synchronization.

- *Acquire-release data synchronization* — if processors $P_1, \ldots, P_k$ need exclusive access to a set of shared locations $\{L_1, \ldots, L_m\}$, the synchronization is accomplished by each processor performing an $acquire(\{L_1, \ldots, L_m\})$ operation followed by a $release(\{L_1, \ldots, L_m\})$ operation [13].

Though it may be possible for some synchronization operation in the above set to be implemented using other operations from the set, it is important that the synchronization operations listed above be identified separately so as to specify the rules for updating the state of a memory location as directly as possible. In particular, we explicitly distinguish between data and control synchronization operations, while many other memory consistency models have focused only on the former [2].

4 Abstraction of Memory System

4.1 Memory Abstraction as Partially Ordered Multisets

The state of a memory location, L, is represented by a *partially ordered multiset* (pomset), $state(L) = (S, \prec)$, where S is a multiset and $\prec \subseteq S \times S$ is a partial order on S [22]. Each element e of the multiset S denotes a write operation or a synchronization operation involving location L. We also define $processorset(e)$ to be the set of processors involved in operation e. Specifically, an element e in multiset S, can denote one of eight different types of operations, as follows:

1. Element $e = write(P_i, v)$ denotes a write operation on location L from processor P_i with value v and $processorset(e) = \{P_i\}$.

2. Element $e = signal(P_i, P_j)$ denotes a signal synchronization operation from processor P_i to processor P_j with $processorset(e) = \{P_i\}$.

3. Element $e = wait(P_i, P_j)$ denotes a wait synchronization operation by processor P_j for processor P_i with $processorset(e) = \{P_j\}$.

   For convenience, we define $signal\_source(e) = e'$, where e' is the *signal* operation that was matched with *wait* operation e at run-time.

4. Element $e = sync(\{P_1, \ldots, P_k\})$ denotes a barrier synchronization operation among processors $P_1, \ldots, P_k$ with $processorset(e) = \{P_1, \ldots, P_k\}$.

5. Element $e = sync\_write(P_i, v)$ denotes a data synchronization write operation on location L from processor P_i with value v and $processorset(e) = \{P_i\}$.

6. Element $e = sync\_read(P_j, v)$ denotes a data synchronization read operation on location L from processor P_i with value v and $processorset(e) = \{P_j\}$.

   For convenience, we define $sync\_write\_source(e) = e'$, where e' is the *sync_write* operation that was matched with *sync_read* operation e at run-time.

7. Element $acquire(P_i)$ denotes an acquire synchronization operation on location L by processor P_i with $processorset(e) = \{P_i\}$.

   For convenience, we define $most\_recent\_release(e) = e'$, where e' is the most recent *release* operation performed on location L prior to acquire action e. Due to the mutual exclusion imposed by acquire/release, there can be at most one such release operation e'. If e is the first acquire operation to be performed on location L, we say that $most\_recent\_release(e)$ is undefined.

8. Element $release(P_i)$ denotes an release synchronization operation on location L by processor P_i with $processorset(e) = \{P_i\}$. For the sake of simplicity, we assume that each consecutive acquire/release pair of operations performed on a given memory location must be issued by the same processor.

4.2 State-Update Operations for a Memory Location

Section 4.1 defined the possible structures for the state (pomset) of a memory location. In this section, we specify how the state (pomset) is updated when a memory or synchronization operation is performed from one of the eight different types identified in section 4.1. The default rule for computing the new pomset $(S_{new}, \prec_{new})$ from the old pomset $(S_{old}, \prec_{old})$ after operation e, is specified as follows using the definition of $processorset(e)$:

$$\begin{aligned} S_{new} &:= S_{old} \cup \{e\} \\ \prec_{new} &:= \prec_{old} \cup \{(x,e) | x \in S_{old} \ \wedge \ processorset(x) \cap processorset(e) \neq \emptyset\} \end{aligned}$$

The default rule is simple. The new operation, e, is inserted into the multiset S, and the partial order $\prec$ is updated so that there is an ordering relation created from any element x in S such that $processorset(e)$ and $processorset(x)$ have a non-empty intersection.

This default rule is applicable to five of the eight different operations that update the state of a memory location (specifically, it is applicable to the *write*, *signal*, *sync*, *sync_write*, and *release* operations). For completeness, we list all eight operations, and provide the state-update rules for the three cases in which the default rule does not apply:

1. *write* — when a write operation from processor P_i to location L with value v is executed, the element $e = write(P_i, v)$ is inserted into the pomset for $state(L)$, using the default state-update rule.

2. *signal* — when a signal operation from processor P_i to processor P_j is executed, the element $e = signal(P_i, P_j)$ is inserted into the pomset for $state(L)$, over *all* memory locations L, using the default state-update rule.

   Note that this synchronization operation and other control-oriented synchronization operations can potentially affect the partial ordering of operations on all memory locations. Therefore, for the sake of completeness, a new element is inserted into the pomset for each memory location.

3. *wait* — when a wait operation from processor P_j to processor P_i is completed, the element $e = wait(P_i, P_j)$ is inserted into the pomsets for *all* locations L.

   Formally, $S_{new} := S_{old} \cup \{e\}$, and $\prec_{new} := \prec_{old} \cup \{(x,e) | x \in S_{old} \ \wedge \ processorset(x) \cap processorset(e) \neq \emptyset\} \cup \{(signal\_source(e), e)\}$.

4. *sync* — when a sync operation among processors $P_1, \ldots, P_k$ is completed, the element $e = sync(\{P_1, \ldots, P_k\})$ is inserted into the pomsets for *all* memory locations L, using the default state-update rule.

5. *sync_write* — when a sync_write operation from processor P_i to location L with value v is executed, the element $e = sync\_write(P_i, v)$ is inserted into the pomset for location L, using the default state-update rule.

6. *sync_read* — when a sync_read operation from processor P_i to location L is executed, the element $e = sync\_read(P_i, v)$ is inserted into the pomset for location L.

   Formally, $S_{new} := S_{old} \cup \{e\}$, and $\prec_{new} := \prec_{old} \cup \{(x, e) | x \in S_{old} \ \wedge \ processorset(x) \cap processorset(e) \neq \emptyset\} \cup \{(sync\_write\_source(e), e)\}$.

7. *acquire* – when an acquire operation is performed on location L by processor P_i, the element $e = acquire(P_i)$ is inserted into the pomset for location L.

   Formally, $S_{new} := S_{old} \cup \{e\}$, and $\prec_{new} := \prec_{old} \cup \{(x, e) | x \in S_{old} \ \wedge \ processorset(x) \cap processorset(e) \neq \emptyset\} \cup \{(e', e) | e' = most\_recent\_release(e)$ is well defined$\}$.

8. *release* – when a release operation is performed on location L by processor P_i, the element $e = release(P_i)$ is inserted into the pomset for location L, using the default state-update rule.

Note that the pomset model for the state of a memory location is solely an abstraction used for defining the LC model. It is not intended that the pomset abstraction and the accompanying state-update rules be actually implemented on the multiprocessor system.

4.3 State Observability for a Memory Location

The rules in section 4.2 specify how the state of a memory location is updated. In this section, we define how the state is observed via *read* operations.

Consider a *read* operation on memory location L from processor P_i. Since the state, $(S, \prec)$, of L is a pomset, in general, the read operation may return one from a set, V, of possible values depending on $(S, \prec)$. We now specify the rule for deriving the value set V from $(S, \prec)$, for a given *read* operation. As defined later in section 5, any hardware implementation that satisfies the LC model model must return some value from V as a result of the read operation.

Definition 4.1 *Element $e = write(P_j, v)$ is said to be a most recent write operation in pomset $(S, \prec)$ for location L if there is no other element $e' \neq e$ such that e' is a write operation and $e \prec e'$.*

We define $MRW(S, \prec)$ to be the set of all elements from pomset $(S, \prec)$ that are most recent write operations. □

Let $(S', \prec')$ be an extended pomset obtained by inserting $e = read(P_i)$ into $(S, \prec)$ as follows:

$$\begin{aligned} S' &= S \cup \{e\} \\ \prec' &= \prec \cup \{(e', e) | e' \in S \wedge P_i \in processorset(e')\} \end{aligned}$$

The value set for read operation $e = read(P_i)$ in $(S', \prec')$ is defined as

$$V(e) = \{val \mid \exists e' = write(P_j, val) \in S' \text{ and } e' \text{ satisfies Condition 1 or 2 listed below}\}$$

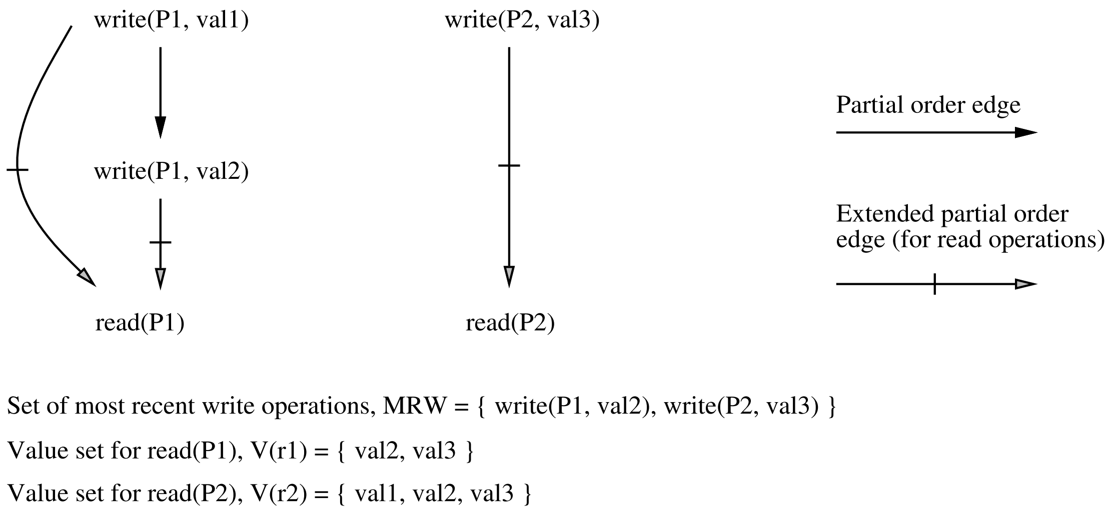

Figure 5: Value sets for read operations r1 and r2 in Figure 3

where conditions 1 and 2 are defined as follows:

- **Condition 1:** $e' = write(P_j, v)$ and $e' \prec' e$ and $e' \in MRW(S, \prec)$
  If e' is a write operation that precedes e in $\prec'$, then it only contributes to V if it is a most recent write operation.

- **Condition 2:** $e' = write(P_j, v)$ and $e' \not\prec' e$
  If e' is a write operation that does not precede e in $\prec'$, then it contributes to V.

As an example, Figure 5 illustrates the value sets defined for read operations $r1$ and $r2$ from the sample program in Figure 3. Assuming that the write operations $w1$, $w2$, $w3$ are scheduled before the read operations $r1$ and $r2$, the pomset for location L will be given by $(S, \prec)$ where $S = \{w1, w2, w3\}$ and $\prec = \{(w1, w2)\}$. This implies that the set of most recent write operations is $MRW(S, \prec) = \{w2, w3\}$. The pomset is then extended with the edge $(w2, r1)$ for read operation $r1$, and with edge $(w3, r2)$ for read operation $r2$; for convenience, we show both extended edges together in Figure 5. The value set for read operation $r1$ is given by $V(r1) = \{val2, val3\}$; $val2$ is included because write operation $w2$ satisfies Condition 1, and $val3$ is included because write operation $w3$ satisfies Condition 2 with respect to read operation $r1$. The value set for read operation $r2$ is given by $V(r2) = \{val1, val2, val3\}$; $val3$ is included because write operation $w3$ satisfies Condition 1, and $val1$ and $val2$ are included because write operations $w1$ and $w2$ satisfy Condition 2 with respect to read operation $r2$. Though it isn't shown in Figure 5, we can similarly obtain the value set for read operation $r3$ as $V(r3) = \{val2, val3\}$, by include an element for the synchronization operation in the pomset for location L.

5 Definition of Location Consistency (LC) model

In this section, we define the Location Consistency (LC) model for memory consistency. The definition uses the concept of an abstract interpreter for a given concurrent program. The abstract interpreter works under an idealized execution model as outlined below.

The execution model for the abstract interpreter maintains the state of the memory location as a pomset. The initial state of the memory location is assumed to be the empty set. The abstract interpreter mimics the execution of the memory and synchronization operations encountered in the concurrent program, and updates the states of memory locations according to the rules specified in section 4.2. For each read operation r on location L, the abstract interpreter computes the value set $V(r)$ from the pomset for location L, and (arbitrarily) returns a value from the set $V(r)$ as the result of the read operation[1]

The execution model of the abstract interpreter makes the following two assumptions:

- **Assumption 1:** As in other memory consistency models [13], we retain the assumption that each processor in the abstract interpreter sequentially executes its node program, subject to all uniprocessor control and data dependences being satisfied.
- **Assumption 2:** It updates the states of memory locations according to the rules specified in section 4.2. For each read operation r on location L, the abstract interpreter computes the value set $V(r)$ from the pomset for location L, and (arbitrarily) returns a value from the set $V(r)$ as the result of the read operation[2]

Analogous to the notion of a *sequential consistent execution* used in defining the data-race-free-1 model in [2], we introduce the notion of a *location consistent execution* as any execution of a program by the abstract interpreter discussed above. Note that, the execution model of the abstract interpreter makes no assumption on the timing of events in the program execution. Therefore, there may be many location consistent executions for a given set of program inputs, due to the nondeterminism that may be inherent in the program and the different timing assumed for the execution of the operations.

The Location Consistency (LC) model is now defined as follows:

Definition 5.1 *A multiprocessor system is location consistent if for any execution of a program on the system, 1) the operations of the execution are the same as those for some location consistent execution of the program, and 2) for each read operation R (with target location L) that is executed on the multiprocessor, the result returned by R belongs to the value-set, V,*

[1] If $V(r) = \emptyset$, the empty set, it means that the memory location has not been initialized, and the abstract interpreter should take whatever action is specified in the program model for a read access to an uninitialized location e.g. return an undefined value, or raise an exception.

[2] If $V(r) = \emptyset$, the empty set, it means that the memory location has not been initialized, and the abstract interpreter should take whatever action is specified in the program model for a read access to an uninitialized location e.g. return an undefined value, or raise an exception.

specified by the state of the memory location L as maintained by the abstract interpreter in the corresponding location consistent execution.

Now let us introduce a sufficient condition for a multiprocessor system to satisfy the location consistency model:

Sufficient Condition for Location Consistency

A hardware system is *location consistent* if the following conditions are satisfied:

1. the operations of the execution are the same as those for some location consistent execution of the program, and

2. for each read operation R (with target location L) that is executed on the multiprocessor, the result returned by R belongs to the value-set, V, specified by the state of the memory location L as maintained by the abstract interpreter in the corresponding location consistent execution.

The validity of the above condition is obvious, as any hardware system that satisfies the sufficient condition should behave like some version of the abstract interpreter.

Our definition of LC only ensure that only ensures that the results of any (every) execution of a program on the location consistent hardware system can be obtained by some location consistent execution of the program, but not vice versa! That is it does not ensures that, the results of any (every) location consistent execution of a program can be obtained by some execution of the program on the location consistent hardware system. Since the results of each read operation in a location consistency execution is defined by a set of possible values, a LC consistent hardware system may be biased to return only one from a subset of the values. Our definition dose not excludes such a hardware system from the class the LC consistent definition.

Therefore, we might also want to introduce the notion of "strongly LC consistent" as follow:

Definition 5.2 *A hardware system is stronly location consistent if the results of any (every) execution of a program on the system can be obtained by some location consistent execution of the program, and vice versa: the results of any (every) location consistent execution of a program can be obtained by some execution of the program on the location consistent hardware system.*

Note that the LC model does not require the memory coherence condition assumed by many other models. The potential for performance improvements as well as other implemntation implications are discussed next in section 6.

6 Performance Potential and Implementation Issues

In this section, we discuss the performance potential of the LC model and its implementation issues.

As mentioned earlier, the LC model does not rely on the memory coherence assumption present in [13] and other SC-derived models. That is, we do not make the assumption that all writes to the same location are serialized in some order and are performed in that order with respect to all processors. Recently, the authors of [13] observed that the memory coherence assumption conflicts with hardware optimizations found in modern processor architectures with agressive instruction-level parallelism e.g. superscalar architectures [12]. For example, one hardware optimization that conflicts with the memory coherence assumption is the use of write buffers [15], in which a processor may retrieve its most recently written value to a given location from its write buffer without involving the memory system. [12] discusses the implementation difficulties that arise when trying to accomodate the memory coherence assumption in conjunction with such hardware optimizations; it also observes that the behavior of existing multiprocessor systems, such as the DASH multiprocessor [20], often does not completely satisfy the requirements of sequential consistency. Since the LC model does not depend on the memory coherence assumption, such complications in implementing hardware optimizations will not exist, hence the potential for higher performance.

Note that writes to the same location are not "serialized" in the LC Model. In fact, the memory state is kept as a partially ordered multiset of memory operations and synchronization operations. The observable order of write operations to the same location may be different for different processors as demonstrated by the examples in Section 2. This flexibility distinguishes the LC model from the SC and SC-derived models. We also believe that the LC model is more general than the SC-derived models, because asynchronous algorithms with data races can be written naturally under this model. However, such programs do not belong to the set of programs intended to be supported by the SC-derived models, e.g. consider the set of properly labeled programs under the RC model. Although, asynchronous algorithms can be written as "improperly-labeled" programs under the SC-derived models, the programmer is left on his own to reason about the concurrent semantics of such programs without getting much help from the memory consistency model.

Now let us examine the implementation requirements for the LC model. One requirement is that the order of memory operations issued from the same processor to the same location must be preserved, when the operations are processed by the memory system. This requirement is already satisfied by the cache-coherence mechanisms and processor-memory interconnects for many existing shared-memory multiprocessor machines. Another requirement is to ensure that pending memory requests are completed before a relevant synchronization operation, so as to enforce the proper partial ordering among memory and synchronization operations. There are proposed solutions in the literature for satisfying this requirement in the presence of *sync_read* and *sync_write* operations [27, 4] and of *acquire* and *release* operations [20]. These solutions would also apply to satisfying the requirements imposed by these synchronization operations under the LC model. Similar solutions will need to be developed for other synchronization operations; for synchronization operations that are implemented in software (as system calls, say), it would be easy to add a "fence"-like instruction to ensure that the requirement is satisfied.

The implementation requirements imposed by the LC model are weaker than the requirements imposed by the SC and SC-derived models. For example, the LC model permits multiple writes from different processors to the same location to be executed concurrently. Consider the sample concurrent program in Figure 3. In the LC model, there is no need for either processor P_1 or P_2 to gain unique ownership of location L before it commits a write. Both processors can process the write operations simultaneously, so long as each subsequent read operation will return some value from its value set as defined by the LC model. Thus, the LC model allows multiple write-ownership of memory locations, a feature that is not permitted by many existing cache coherence mechanisms. We believe that this flexibility offered by the LC model will simplify the implementation of cache coherence support and provide the potential for additional performance improvements. For example, the Data Merging protocol presented in [17] can be used to provide efficient cache coherence support under the LC model, but not under the SC and SC-derived models.

7 Relation with Other Memory Consistency Models

In this section, we discuss related work in the area of memory consistency models. Our discussion is focused on the SC and SC-derived models. We have already discussed some comparisons between the LC model and the SC and SC-derived models in earlier sections of this paper. The purpose of this section is is to provide more detailed background information on the related work.

The notion of *sequential consistency* was originally introduced by Lamport [19] as follows:

> Sequential Consistency (SC): A system is sequentially conistent if the result of any execution is the same as if the operations of all the processors were executed in some sequential order, and the operations of each individual processor appear in this sequence in the order specified by its program.

As discussed earlier, the SC model limits performance by preventing the use of common uniprocessor hardware optimizations such as store buffers and out-of-order memory operations [13, 1]. To relax the ordering constraints on memory operations, while still maintaining sequential consistency, a number of weaker memory consistency model were proposed.

The *weak consistency* (WC) model has been proposed by Dubois, Sceurich and Briggs [10]. Under the WC model, ordinary shared memory access operations and synchronization memory access operations are distinguished. The synchronization access operations are used to control the interaction among concurrent processes to maintain the consistency of the shared memory. Hence, in WC model, only synchronization access operations are ordered, and the ordering obeys the SC model. For ordinary shared memory accesses, mutual exclusion should be ensured by using constructs such as critical sections implemented by synchronization memory operations.

To further relax the ordering constraints, Gharchorloo et al have proposed the release consistency (RC) model [13]. The RC model exploits additional information about shared access operations. In particular, synchronization access operations are partitioned into *acquire* and *release* operations. An acquire synchronization access operation is used to gain access permision to a set of shared locations — like a lock operation — before the issuing process performs ordinary memory operations on these shared locations. A release synchronization access operation is used to return the access permision. Acquire and release operations usually appear in pairs. A class of programs, formally identified as *properly labeled programs*, makes judicious use of the acquire and release operations so as to ensure that sequential consistency is maintained.

There are many variations of the three memory consistency models discussed so far (SC, WC and RC). For example, Goodman has introduced the *processor consistency* (PC) model to relax the odering constraints imposed by SC [14]. Under the PC model, the order in which write accesses from two processors (to different locations) occur may not be the same as observed by themselves or by a third processor. Read accesses may bypass write accesses in a processor – an architectural optimization supported by many today's processor architectures. Another example is the *lazy release consistency* (LRC) model has been proposed by Keleher et. al [18]. The objective of LRC is to reduce both the number of messages and the amount of data transferred when implementing the RC model. A major feature of LRC is that it does not make modifications globally visible at the time of a release operation is performed. Instead, it gurantees only that a processor that acquires a lock will see all modifications that precede the lock acquire.

A central assumption made by the SC and SC-derived mdels is that sequential consistency is preferred by most programmers since it appears to be a straightforward extension of the uniprocessor memory abstraction to multiprocessor systems. There are two important features common to the definition all these SC-derived memory consistency models:

1. The assumption of *memory coherence*: the SC-derived models assume that the memory is kept coherent, that is, "all writes to the same location are serialized in some order and are performed in that order with respect to any processor" [13].

2. The abstraction of memory access operations: the SC-derived models use the concepts of a memory access being *performed* or *globally performed*. These concepts are defined in [13].

The notions of "perfomed" and "globally performed" are based on Dubois' memory abstraction as summarized in the following definitions from [13]:

Definition 1: A LOAD by a processor P_i is considered *performed* with respect to a processor P_k **at a point in time** when the issuing of a STORE to the same address by P_k cannot affect the value returned by the LOAD. A STORE by a processor P_i is considered *performed* with respect to a processor P_k **at a point in time** when an issued LOAD to the same address by P_k returns the value defined by this STORE (or a subsequent STORE to the same location). An access is *performed* when it is performed with respect to all processors.

And the following definition describes the notion of "globally performed":

> Definition 2: A LOAD is *globally performed* if it is performed and if the STORE that is the source of the returned value has been performed.

Two important differences between the LC model and the SC-derived models are that the LC model relies neither on the assumption of memory coherenc nor. on the notions of "performed" and "globally performed".

Now let us come back to the notion of "performed" and "globally performed" which is based on the concept of "at a point in time". These notions implicitly rely on the assumption of some form of global time which, we argue, is hard to interpret and maintain in a multiprocessor system. As pointed out by Stone [28], the rules for defining these terms are quite "vague", and it is hard to pin down just what is the "point of time" used in the definition, in particular when dealing with multiprocessors with coherent cache systems and buffered memory-processor interconnection networks..

TheLC model has no notion of global time. The control of memory state is fully distributed across individual memory locations, and is governed by a set of straightforward rules. As a result, the LC model provides a simpler contract between software and hardware, compared to the SC and SC-derived models.

Today, it is largely a matter of subjective opinion as to whether or not programmers will prefer the LC model over the other SC and SC-derived models. We should emphasize, however, that the purpose of introducing the LC model is not to contradict other memory consistency models. Instead, the LC model should be viewed as an extension of existing models by further relaxing the ordering constraints imposed by the memory coherence assumption.

Due to space limitations, we will not enumerate many other interesting memory consistency models, such as PRAM [21], the total memory ordering [29], the partial memory ordering [29], the slow memory model [16], the casual memory model [16], and the concurrent consistency model [25]. A brief survey of these models can be found in [2].

8 Conclusions and Future Work

Memory consistency models play an important role on the correct and efficient implementation of shared memory multiprocessor systems. A memory consistency model defines the orders in which memory accesses can be executed by the hardware and be observed by the software.

In this paper we have developed the location consistency (LC) model. An important feature of the LC model is that it does not rely on the memory coherence assumption, unlike other SC-derived models. The LC model is based on a memory abstraction in which each memory location is modeled by a pomset, thus memory operations on the same location need not be serialized as in the SC-derived models. The LC model does not impose any unnecessary ordering of memory operations beyond what is defined by the program's partial order. We have presented the advantages of the LC model in terms of performance potential as well as of its implementation implications.

The LC model provides a new framework for memory consistency models beyond the SC-derived models. As future work, we plan to exploit the power of the LC model by designing/extending appropriate compiler optimizations permited by the LC model and by designing new, efficient, and perhaps more aggressive, implementations of memory systems (including caches) in shared memory multiprocessors.

References

[1] Sarita V. Adve and Mark D. Hill. Weak ordering—a new definition. In *Proceedings of the 17th Annual International Symposium on Computer Architecture*, pages 2–14, Seattle, Washington, May 28–31, 1990. IEEE Computer Society and ACM SIGARCH. *Computer Architecture News,* 18(2), June 1990.

[2] Sarita V. Adve and Mark D. Hill. A unified formalization of four shared-memory models. *IEEE Transactions on Parallel and Distributed Systems*, pages 613–624, June 1993.

[3] Alfred V. Aho, Ravi Sethi, and Jeffrey D. Ullman. *Compilers — Principles, Techniques, and Tools.* Addison-Wesley Publishing Company, Reading, Massachusetts, 1986.

[4] Arvind, M. L. Dertouzos, R. S. Nikhil, and G. M. Papadopoulos. Project dataflow—the Monsoon architecture and the Id programming language. CSG Memo 285, Computation Structures Group, MIT Laboratory for Computer Science, Cambridge, Massachusetts, March 1988.

[5] P. Brinch Hansen. The programming language Concurrent Pascal. *IEEE Transactions on Software Engineering*, 1(2):199–206, June 1975.

[6] F. Darema, D. A. George, V. A. Norton, and G. F. Pfister. A single-program-multiple-data computational model for EPEX/FORTRAN. *Parallel Computing*, 7:11–24, April 1988.

[7] R. De Leone and O. L. Mangasarian. Asynchronous parallel successive overrelaxation for the symmetric linear complementarity problem. *Mathematical Programming*, 42(2):347–361, November 1988.

[8] E. W. Dijkstra. Co-operating sequential processes. In F. Genuys, editor, *Programming Languages*, pages 43–112. New York: Academic Press, 1968.

[9] Anne Dinning and Edith Schonberg. An empirical comparison of monitoring algorithms for access anomaly detection. In *Proceedings of the Second ACM SIGPLAN Symposium on Principles & Practice of Parallel Programming (PPOPP)*, pages 1–10, Seattle, Washington, March 14–16, 1990. *SIGPLAN Notices,* 25(3), March 1990.

[10] Michel Dubois, Christoph Scheurich, and Faye Briggs. Memory access buffering in multiprocessors. In *Proceedings of the 13th Annual International Symposium on Computer Architecture*, pages 434–442, Tokyo, Japan, June 2–5, 1986. IEEE Computer Society and ACM SIGARCH. *Computer Architecture News,* 14(2), June 1986.

[11] Jeanne Ferrante, Karl J. Ottenstein, and Joe D. Warren. The program dependence graph and its use in optimization. *ACM Transactions on Programming Languages and Systems*, 9(3):319–349, July 1987.

[12] Kourosh Gharachorloo, Anoop Gupta, and John Hennessy. Revision to 'memory consistency and event ordering in scalable shared-memory multiprocessors'. Technical Report No. CSL-TR-93-568, Computer Systems Laboratory, Stanford University, Stanford, California, April 1993.

[13] Kourosh Gharachorloo, Daniel Lenoski, James Laudon, Phillip Gibbons, Anoop Gupta, and John Hennessy. Memory consistency and event ordering in scalable shared-memory multiprocessors. In *Proceedings of the 17th Annual International Symposium on Computer Architecture*, pages 15–26, Seattle, Washington, May 28–31, 1990. IEEE Computer Society and ACM SIGARCH. *Computer Architecture News,* 18(2), June 1990.

[14] J. R. Goodman. Cache consistency and sequential consistency. Technical Report 1006, Department of Computer Science, University of Wisconsin, Madison, February 1991.

[15] John L. Hennessy and David A. Patterson. *Computer Architecture: A Quantitative Approach.* Morgan Kaufmann Publishers, Inc., 1990.

[16] P. W. Hutto and M. Ahamad. Slow memory: Weakening consistency to enhance concurrency in distributed shared memories. In *Proceedings of the 10th International Conference on Distributed Computing Systems*, pages 302–311, Paris, France, May 28–June 1, 1990. IEEE Computer Society.

[17] Alan H. Karp and Vivek Sarkar. Data merging for shared-memory multiprocessors. In *Proceedings of the Twenty-Sixth Annual Hawaii International Conference on System Sciences*, volume II, pages 244–256, Koaloa, Hawaii, January 5–8, 1993. ACM and IEEE Computer Society. Also published as Technical Report 03.454, IBM Santa Teresa Laboratory, October 1992.

[18] Pete Keleher, Alan L. Cox, and Willy Zwaenepoel. Lazy release consistency for software distributed shared memory. In *Proceedings of the 19th Annual International Symposium on Computer Architecture*, pages 13–21, Gold Coast, Australia, May 19–21, 1992. ACM SIGARCH and IEEE Computer Society. *Computer Architecture News,* 20(2), May 1992.

[19] Leslie Lamport. How to make a multiprocessor computer that correctly executes multiprocess programs. *IEEE Transactions on Computers*, 28(9):690–691, September 1979.

[20] Daniel Lenoski, James Laudon, Truman Joe, David Nakahira, Luis Stevens, Anoop Gupta, and John Hennessy. The DASH prototype: Implementation and performance. In *Proceedings of the 19th Annual International Symposium on Computer Architecture*, pages 92–103, Gold Coast, Australia, May 19–21, 1992. ACM SIGARCH and IEEE Computer Society. *Computer Architecture News,* 20(2), May 1992.

[21] R. J. Lipton and J. S. Sandberg. PRAM: a scalable shared memory. Technical Report CS-TR 180–88, Princeton University, September 1988.

[22] V. R. Pratt. The duality of time and information. In *Proceedings of the Third International Conference on Concurrency Theory*, pages 237–253. Springer-Verlag, August 1992.

[23] V. Sarkar. A concurrent execution semantics for parallel program graphs and program dependence graphs. In Uptal Banerjee, David Gelernter, Alex Nicolau, and David Padua, editors, *Proceedings of the 5th International Workshop on Languages and Compilers for Parallel Computing*, number 757 in Lecture Notes in Computer Science, pages 16–30, New Haven, Connecticut, August 3–5, 1992. Springer-Verlag. Published in 1993.

[24] Vivek Sarkar. Synchronization using counting semaphores. In *Conference Proceedings, 1988 International Conference on Supercomputing*, pages 627–637, St. Malo, France, July 4–8, 1988. ACM.

[25] C. E. Scheurich. *Access Ordering and Coherence in Shared Memory Multiprocessors.* PhD thesis, University of Southern California, 1989.

[26] A. C. Shaw. *The Logical Design of Operating Systems.* Prentice-Hall, Inc., 1974.

[27] Burton Smith. The architecture of HEP. In Janusz S. Kowalik, editor, *Parallel MIMD Computation: HEP Supercomputer and its Applications*, Scientific Computation Series, pages 41–55. MIT Press, Cambridge, Massachusetts, 1985.

[28] Harold S. Stone. *High-Performance Computer Architecture.* Addison-Wesley Publishing Company, 3rd edition, 1993.

[29] Sun Microsystems Inc. *The SPARC Architecture Manual*, 1991.