University of Delaware
Department of Electrical and Computer Engineering
Computer Architecture and Parallel Systems Laboratory

Location Consistency – a New Memory Model and Cache Consistency Protocol

Guang R. Gao
Vivek Sarkar †

CAPSL Technical Memo 16
February 16, 1998

†IBM Thomas J. Watson Research Center, P.O.Box 704, Yorktown Heights, NY 10598, USA, Email:vivek@watson.ibm.com

University of Delaware • 140 Evans Hall • Newark, Delaware 19716 • USA
http://www.capsl.udel.edu • ftp://ftp.capsl.udel.edu • capsladm@capsl.udel.edu

Abstract

Existing memory models and cache consistency protocols assume the *memory coherence* property which requires that all processors observe the same ordering of write operations to the same location. In this paper, we address the problem of defining a memory model that does not rely on the memory coherence assumption, and also the problem of designing a cache consistency protocol based on such a memory model. We define a new memory consistency model, called Location Consistency (LC), in which the state of a memory location is modeled as a partially ordered multiset (pomset) of write and synchronization operations. We prove that LC is strictly weaker than existing memory models, but is still equivalent to stronger models for parallel programs that have no data races.

We also introduce a new multiprocessor cache consistency protocol based on the LC memory model. We prove that this LC protocol obeys the LC memory model. The LC protocol does not need to enforce single write ownership of memory blocks. As a result, the LC protocol is simpler and more scalable than existing snooping and directory-based cache consistency protocols.

Contents

List of Figures

1 Introduction

A memory consistency model specifies the semantics of concurrent memory operations, such as load/store and synchronization operations, in a multiprocessor system. The most commonly used hardware memory consistency model is *sequential consistency* (SC) [17]. The main approach taken in recent work on memory consistency models is to allow performance optimizations to be applied, while ensuring that sequential consistency is retained for a restricted class of programs — mainly programs that do not exhibit data races [1][1]. Therefore, we refer to these weaker memory consistency models as *SC-derived* models.

A fundamental limitation of all SC-derived memory consistency models is the *memory coherence* assumption, which can be stated as follows [12]: "all writes to the same location are serialized in some order and are performed in that order with respect to any processor". We have argued earlier that the memory coherence assumption poses fundamental obstacles to defining a scalable and efficient view of memory consistency in computer systems [11]. The memory coherence assumption imposes restrictions on the ordering of memory operations that go beyond the partial order defined by synchronization operations in a parallel program. In this paper, we address the problem of defining a memory consistency model and of designing an accompanying cache consistency protocol based on a partial order semantics that does not rely on the memory coherence assumption.

The rest of the paper is organized as follows. Section 2 introduces the LC memory model. Section 3 establishes some important properties of the LC model to demonstrate its usefulness. Section 4 describes the LC cache consistency protocol. Section 5 discusses related work and section 6 contains our conclusions.

2 The Location Consistency (LC) Model

The main purpose of a memory model is to serve as an agreement between hardware and software on the semantics of memory operations so as to ensure correct execution of user programs. However, the bulk of past work on memory models has been pursued from the hardware viewpoint. These models assume the *memory coherence* property because it is viewed as a "natural" property from the hardware viewpoint. Our position is that it is essential to adopt an end-to-end view of memory consistency to obtain memory models that are scalable and that can be understood uniformly at all levels of software and hardware. We believe that this is possible with a memory consistency model based on partial order execution semantics, which is the motivation for the Location Consistency (LC) model described in this section.

[1] In a data-race-free parallel program, all accesses to shared data must be protected by data synchronization or by control sequencing.

2.1 Program Model

In this section, we outline the program model assumed for this work. This model is constructed in such a way as not 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 [21], cobegin-coend [4], Single Program Multiple Data (SPMD) streams [6], program dependence graphs [7], and parallel program graphs [20].

The primitive memory and synchronization operations assumed in defining the LC model are as follows. For simplicity, we assume that all other shared-memory and synchronization operations in a multiprocessor system can be defined as a combination of these memory and synchronization operations. A definition of the LC model for a richer set of memory and synchronization operations can be found in [9, 10].

- *Memory write* — if processor P_i needs 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 needs 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.
- *Acquire-release data synchronization* — if processors $P_1, \ldots, P_k$ need exclusive access to a shared location L, the synchronization is accomplished by each processor performing an $acquire(P_i, L)$ operation followed by a $release(P_i, L)$ operation [12]. In general, the processors may use acquire-release to request exclusive access to a set of shared variables, $\{L_1, \ldots, L_m\}$, rather than a single location.

2.2 LC Memory Consistency Model

We now define the LC memory model for the program model outlined in section 2.1. Definitions of the LC memory model for more complicated program models were provided in [9, 10].

Abstraction of Memory System

In Location Consistency, we view the state of a memory location as a partial order (rather than a total order) of write operations. Specifically, the state of a memory location, L, is conceptually 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. Each element e of the multiset S denotes a write operation or a synchronization operation involving location L. We define *processorset*(e) to be the set of processors involved in operation e. For the operations defined in section 2.1, *processorset(e)* will always be a singleton; however, *processorset(e)* can be a set in general when considering operations such as barrier synchronization as discussed in [9, 10].

Specifically, an element e in multiset S can denote one of three different types of operations:

- Element $e = write(P_i, v, L)$ denotes a write operation on location L from processor P_i with value v and $processorset(e) = \{P_i\}$.
- Element $e = acquire(P_i, L)$ denotes an acquire synchronization operation on location L by processor P_i with $processorset(e) = \{P_i\}$.
- Element $e = release(P_i, L)$ 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.

For a given acquire operation e, 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.

State-Update Operations for a Memory Location

The state (pomset) for a memory location is updated when a *write*, *acquire*, or *release* operation is performed. The basic rule for computing the new pomset $(S_{new}, \prec_{new})$ from the old pomset $(S_{old}, \prec_{old})$ after operation e, is as follows:

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

The basic rule is simple. The new operation, e, is inserted into the multiset S, and the partial order $\prec$ is updated so that x precedes e in S, if *processorset*(e) and *processorset*(x) have a non-empty intersection. The initial state of the memory location is assumed to be the empty pomset. In general, the partial order captures the sequencing constraints of the memory and synchronization operations performed on the location. There are no additional processor-based sequencing constraints as in the memory coherence assumption. Since all ordering constraints are captured at the location level, we chose the name Location Consistency for our memory model.

The basic update rule defined above is used for *write* and *release* operations. A modified rule with *most_recent_release()* is used for the *acquire* operation. The state update rules for the *write*, *acquire*, and *release* operations are as follows. (State update rules for additional operations can be found in [9, 10].)

- *write* — when a write operation from any 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 basic state-update rule.

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

- *acquire* – when an acquire operation is performed on location L by any processor P_i, the element $e = acquire(P_i)$ is inserted into the pomset for location L. If $e' = most\_recent\_release(e)$ is well defined then e' must have been inserted into the pomset for *state(L)* prior to the insertion of operation e. However, since e and e' may have disjoint *processorset* values, the basic rule has to be augmented to enforce a dependence from e' to e and to obtain a new partial order as follows:

$$\begin{aligned} \prec_{new} := \prec_{old} \;&\cup\; \{(x, e) \mid x \in S_{old} \wedge processorset(x) \cap processorset(e) \neq \emptyset\} \\ &\cup\; \{(e', e) \mid e' = most\_recent\_release(e) \text{ is well defined}\} \end{aligned}$$

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 and the accompanying state-update rules be actually implemented on a real multiprocessor system. Instead, as described below, the pomset defines the set of all permissible values that can be returned for a *read* operation executed on a computer system that satisfies the LC model.

State Observability for a Memory Location

The state-update rules specify how the state of a memory location is updated by write operations and synchronization operations. We now define how the state of a memory location is observed via read operations.

Consider a *read* operation e 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 an element from the set, $V(e)$, of possible "live" values depending on $(S, \prec)$. We now specify the rule for deriving the *value set* $V(e)$ from $(S, \prec)$, for a given read operation.

First, we define an *extended pomset* $(S', \prec')$ obtained by inserting $e = read(P_i, L)$ into $(S, \prec)$ as follows:

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

Now, the value set $V(e)$ for read operation $e = read(P_i, L)$ in $(S', \prec')$ can be defined as follows,

$$\begin{aligned} V(e) = \{\, v \mid\; &\exists\, w = write(P_j, v, L) \in S' \\ &\text{and } w \text{ satisfies Condition 1 or 2 listed below}\} \end{aligned}$$

where conditions 1 and 2 are:

- **Condition 1:** $w = write(P_j, v, L)$ and $w \prec' e$ and $w \in MRPW(S, \prec, e)$
  If w is a write operation that precedes e in $\prec'$, then it can only be included in $V(e)$ if it is a *most recent predecessor write (MRPW)* operation with respect to read operation e. (The *MRPW* set is defined below.)

- **Condition 2:** $w = write(P_j, v, L)$ and $w \not\prec' e$
  If w is a write operation that does not precede e in $\prec'$, then it is automatically included in $V(e)$ (even though w may precede other write operations to location L in $\prec'$).

The *MRPW* set used in condition 1 is defined as follows. Element $w = write(P_j, v, L)$ is said to be a *most recent predecessor write* operation in extended pomset $(S', \prec')$ for location L with respect to read operation e if w is a predecessor of e ($w \prec' e$) and there is no other write operation on location L, $w' \neq w$, such that w' precedes e ($w' \prec' e$) and w precedes w' ($w \prec w'$). (Both $\prec$ and $\prec'$ are assumed to be transitive because they are partial orders.) Thus, $MRPW(S, \prec, e)$ is the set of all write operations from pomset $(S', \prec')$ that are most recent predecessor write operations with respect to read operation e.

As an example, Figure 1 shows the extended partial orders and value sets for two scenarios for the execution of the read operation on processor 1 in the simple parallel program trace shown in figure 1:

Case 1: When the read executes before the acquire on processor 2.

In this case, the partial order only contains a single write operation (from processor 1), and the value set for the read operation is $V(e) = \{\ val1\ \}$ according to condition 1 above.

Case 2: When the read executes after the release on processor 2.

In this case, the partial order contains two write operations (from processors 1 and 2). In the extended partial order, we see that $write(P_1)$ precedes $read(P_1)$ but $write(P_2)$ and $read(P_1)$ are unrelated. Therefore, the *MRPW* set is the same as in case 1. However, the value set for the read operation is $V(e) = \{\ val1,\ val2\ \}$. *val2* is included in $V(e)$ because of condition 2 above.

Since $V(e)$ contains multiple elements, it indicates that there is a data race in the program trace in figure 1. The programmer could have avoided the data race by enclosing the read operation in a pair of acquire-release synchronization operations.

We now formalize the connection between execution scenarios and value sets by precisely defining what it means for a parallel execution of a program to be location consistent.

Definition of Location Consistency Memory Model

Sample parallel program trace: (Only operations involving location X are shown)

```
Processor 1          Processor 2

  . . .                . . .
  acquire(X);          . . .
  X = val1;            . . .
  release(X);          . . .
  . . .                acquire(X);
  . . .                X = val2;
  . . .                release(X);
  read X;              . . .
```

Extended partial orders for two execution scenarios for read operation:

Case 1: read executes before acquire on processor 2.

Case 2: read executes after release on processor 2.

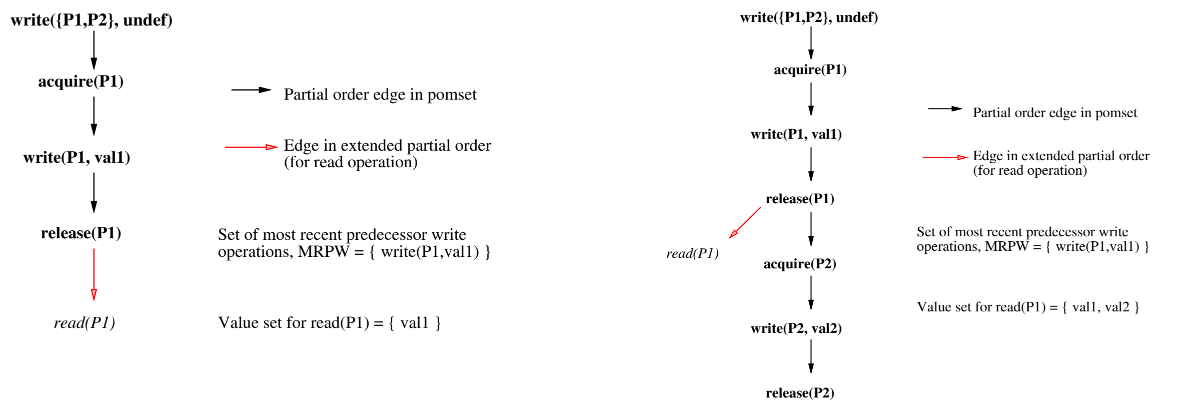

Figure 1: Example of extended partial orders and value sets for read operation

The formal definition for Location Consistency 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 reason for using the abstract interpreter in defining the memory model is to break the cycle between program execution and memory semantics viz., the memory semantics depends on the program execution and the unfolding of the program execution depends on the memory semantics. Many memory models defined in past work ignored this cyclic dependence and assumed the availability of some specific program execution trace when defining the memory semantics.

The execution model for the abstract interpreter maintains the state of each memory location as a pomset. The initial state of each memory location is assumed to be a singleton pomset containing $write(\{P_1, P_2, ...\}, \perp)$ *i.e.,* each memory location is initialized with a (pseudo) write of an undefined value (represented by $\perp$) from all processors at the start of program execution.

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 earlier. 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]. As in definitions of other memory consistency models [12], we assume that all uniprocessor control and data dependences are satisfied.

Analogous to the notion of a *sequential consistent execution* defined in [1], 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 and data races that may be inherent in the program.

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

Definition 2.1. *A multiprocessor system is* **location consistent** *if for any execution of a program on the system:*

- *the operations of the execution are the same as those for some location consistent execution of the program; and*
- *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, where V is specified by the state of the memory location L as maintained by the abstract interpreter in the corresponding location consistent execution.*

Note that our definition of LC requires that the results of any (every) execution of a program on location consistent hardware can be obtained by some location consistent execution of the program on the abstract interpreter, but not vice versa *i.e.*, as might be expected, it does not require that the results of any (every) location consistent execution of a program on the abstract interpreter be reproducible by some execution of the program on location consistent hardware.

3 Properties of the LC Model compared to other Cache Consistency Models

In this section, we establish some important properties of the LC model to demonstrate that it is a robust and reasonable memory consistency model. In addition, we show that the LC

[2] If the value returned is $\perp$, it means that the read operation accessed the undefined initial value of the memory location, 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.

model is strictly weaker than other memory consistency model that have been presented in past work, thus making the LC model more attractive for use in future scalable shared-memory multiprocessor systems. Our study of these properties for the LC model was inspired in part by a study of these kinds of properties in [8] for the dag consistency model using a computation-centric framework *i.e.,* a framework in which memory consistency is defined for some specific (partially ordered) trace of a program execution (unlike our framework based on the abstract interpreter described at the end of section 2.2). Specifically, we prove that the LC model satisfies the four properties listed below:

Weakness Property The LC model is strictly weaker than the Release Consistency (RC) models (which is in turn strictly weaker than the Sequential Consistency model).

The Weakness Property reveals why we can expect to find a more efficient protocol for the LC memory model compared to protocols for the RC and stronger models.

Equivalence Property For parallel programs that have no data races, the LC model is equivalent to the RC model.

Since most parallel programs are designed to be free of data races, the Equivalence Property shows that the LC model is as useful as the RC model (and the SC model) for this large class of programs.

Monotonicity The LC model is monotonic with respect to parallelism — if the LC model permits a certain mapping of values to dynamic instances of read instructions in the execution of a parallel program, it must permit the same mapping in an isomorphic execution of a legal "more parallel" version of the same program (*i.e.,* in an execution that has fewer edges/constraints in the partial order of the program's execution).

Monotonicity is a measure of robustness of the LC model. If a legal parallelization transformation is performed on a program, any set of values returned by read operations in the original program should also be permitted in the parallelized program.

Non-intrusive Reads Reads are non-intrusive in the LC model — the addition or removal of a read instruction in a parallel program cannot change the legality of values returned by dynamic instances of read instructions in a given execution of the parallel program.

Non-intrusive reads is another measure of robustness of the LC model. It implies that the addition or removal of debug statements (for example) will not change the memory semantics of locations being examined.

We will use the notation $\mathcal{E}$ to refer to a specific execution of a parallel program and the notation $\mathcal{R}(\mathcal{E})$ to refer to the *read mapping* for execution $\mathcal{E}$. Specifically, $\mathcal{R}(\mathcal{E})[r] = v$ contains the value v that was returned by (dynamic) read operation r in execution $\mathcal{E}$. In general, $\mathcal{E}$ may refer to a partial execution of a parallel program.

Recall that when the abstract interpreter defined in section 2 performs a location consistent execution of a program, it simulates an abstract memory system in which it maintains the state of each memory location L as a pomset of write/synchronization instructions, $state(L) = (S, \prec)$. The *value set*, $V(r, L)$, for a read operation r on location L can then be derived from $state(L)$, as defined in section 2. In a location consistent execution, each read operation r on some location L must return a value that is an element of $V(r, L)$.

3.1 Weakness Property

In this section, we prove that the LC model is strictly weaker than the RC model (which in turn is strictly weaker than the SC model). The first part of the proof is to show that every release consistent execution of a parallel program also satisfies the LC model. This can be shown by establishing that the value returned for each read operation in an RC execution must also be an element of the *value set* for the read operation in an LC execution. We omit the details for the sake of brevity. The second part of the proof is to show that the LC model is *strictly* weaker than the RC model. To do so, we outline a program execution that satisfies the LC model but does not satisfy the RC model.

Consider a parallel execution $\mathcal{E}$ of the program shown in Figure 2 in which processor 1 writes $L := 1$ asynchronously (without locking), and processor 2 acquires the lock for location L to write $L := 2$. Next, processor 1 acquires the lock for location L and reads the value in L (read operation $r1$); processor 1 later performs a second read operation ($r2$) on location L but this second read is asynchronous *i.e.,* read operation $r2$ is performed without any locking. In the RC model, it is illegal for read operations $r1$ and $r2$ to return different values for program execution $\mathcal{E}$. This is because both write operations, $w1$ and $w2$, are executed in $\mathcal{E}$ *before* read operations $r1$ and $r2$. According to the RC model (and any other consistency model that relies on the memory coherence assumption), all writes to the same location are serialized in some order and are performed in that order with respect to any processor; this implies that read operations $r1$ and $r2$ must both see the same value (either from $w1$ or from $w2$ which ever appears last in the order) in a memory model that obeys memory coherence. Even from this simple example, one can surmise that a cache consistency protocol for the RC model must include some bookkeeping overhead to identify a fixed (serialized) order for all writes to the same location. This bookkeeping is usually achieved by maintaining a single "home location" for each shared location in shared-memory multiprocessors.

However, in the LC model, the value sets for read operations $r1$ and $r2$ are $V(r1) = V(r2) = \{1, 2\}$ *i.e.,* the value sets include the values from both $w1$ and $w2$ which is consistent with the program's partial order relationship revealed by execution $\mathcal{E}$. Thus, read operations $r1$ and $r2$ may return different values under the LC model, say 1 and 2 respectively as shown in figure 2. From this simple example, one can see that cache consistency protocols for the LC model need not incur the bookkeeping overhead of identifying a fixed (serialized) order for all writes to the same location. Instead, a protocol for the LC model can return any value for a read operation r that belongs to the value set $V(r)$. This property is exploited by the LC cache consistency protocol presented in section 4.

```
Processor 1                                          Processor 2

w1:  L := 1

                                                     /* Assume that processor 2 acquires
                                                        location L before processor 1
                                                        in this program execution. */

                                                        acquire(L)
                                                     w2:  L := 2
                                                        release(L)

     acquire(L)
/* Assume that read r1 returns L = 1
   in this program execution. */
r1:  A := L
     release(L)

/* Assume that read r2 returns L = 2
   in this program execution. */
r2:  B := L
```

Figure 2: Example of a parallel program execution that satisfies the LC model but not the RC model

3.2 Equivalence Property

The Weakness Property from section 3.1 reveals why the LC model can be more attractive than other memory consistency models for use in future scalable shared-memory multiprocessor systems. In this section, we show why LC is a useful model by proving that the LC model is equivalent to the RC model for all program executions that are free of data races (access anomalies). Prior work [12] has shown that the RC model is equivalent to the SC model for such program executions, so the Equivalence Property also implies that the LC model is equivalent to Sequential Consistency for all program executions that are free of data races.

The Equivalence Property holds for parallel programs that are free of *data races*. We say that a program is free of data race if all its accesses to shared data are protected by control sequencing or by either direct or indirect data synchronization. This class of programs have also been referred to as *data-race-free* [1] and as *properly labeled* [12] in the literature.

To prove the Equivalence Property, we define a Memory Coherent Abstract Interpreter (MCAI) that is a variant of the Location Consistent Abstract Interpreter (LCAI) introduced in section 2. The MCAI treats each write operation w on location L as having $\mathit{processorset}(w) = \{1, \ldots, P\}$, i.e., every processor is included in $\mathit{processorset}(w)$ regardless of which processor performed the write operation. Therefore, all write operations will be totally ordered in the state of each memory location, instead of the partial order maintained by the LCAI. Further, if we consider an extended pomset $(S', \prec')$ for any read operation e on location L, all write operations to location L must be predecessors of e. Hence, the value set for any location in the MCAI will always have size ≤ 1, because there will be at most one "most recent predecessor write" for each location. The value returned by a read operation in the MCAI will be undefined if the value set is empty. An execution of a parallel program will be release consistent (*i.e.,* will obey the RC model) if the result of each read operation is the same as the result of the read operation in a corresponding execution of the MCAI.

Now consider a location consistent execution $\mathcal{E}$ that has no data races. Since the execution obeys the LC model, the result of each read operation r in $\mathcal{E}$ must belong to the value set $V(r)$ in the LCAI execution that corresponds to $\mathcal{E}$. Since $\mathcal{E}$ has no data races, each $V(r)$ set must have size *leq* 1. Therefore, the value returned by read operation r in the LCAI execution must be identical to the value returned by a corresponding MCAI execution. Hence execution $\mathcal{E}$ obeys the RC model.

3.3 Monotonicity

In this section, we show that the LC model is *monotonic* with respect to parallelism. Consider a location consistent execution $\mathcal{E}_1$ of a given parallel program $\mathcal{P}_1$. Now consider parallel program $\mathcal{P}_2$ obtained by removing a single pair of acquire-release operations from $\mathcal{P}_1$; we say that program $\mathcal{P}_2$ is strictly "more parallel" than program $\mathcal{P}_1$. We assume that this transformation is legal *i.e.,* performing the transformation does not violate any data dependences. (Analogous transformations can be defined for other synchronization operations.)

We derive an execution $\mathcal{E}_2$ for program $\mathcal{P}_2$ from execution $\mathcal{E}_1$ by simply deleting all dynamic instances of the acquire operation and release operations that were deleted from $\mathcal{P}_1$ to obtain $\mathcal{P}_2$. Monotonicity guarantees that $\mathcal{E}_2$ will be a location consistent execution of program $\mathcal{P}_2$.

The proof of this result hinges on a simple observation. Let $\mathcal{R}(\mathcal{E}_1)$ and $\mathcal{R}(\mathcal{E}_2)$ be the read mappings for program executions $\mathcal{E}_1$ and $\mathcal{E}_2$. Consider a read operation r on location L in execution $\mathcal{E}_1$; r must also be present in execution $\mathcal{E}_2$. Let $V_1(r, L)$ and $V_2(r, L)$ be the value sets for read operation r in executions $\mathcal{E}_1$ and $\mathcal{E}_2$ respectively. Since program $\mathcal{P}_2$ is more parallel than program $\mathcal{P}_1$, it must be the case that the partial order $\prec_2$ for the state of location L in execution $\mathcal{E}_2$ is a subset of the partial order $\prec_1$ for the state of location L in execution $\mathcal{E}_1$ *i.e.,* $\prec_2 \ \subseteq \ \prec_1$. This implies a subset relationship among the most-recent-predecessor-write sets with respect to any read operation e on location L, $MRPW_1(S, \prec_1, e) \subseteq MRW_2(S, \prec_2, e)$, and hence among value sets, $V_1(r, L) \subseteq V_2(r, L)$. The value $v = \mathcal{R}(\mathcal{E}_1)[r]$ returned by read operation r in execution $\mathcal{E}_1$ must satisfy $v \in V_1(r, L)$. Therefore, $v \in V_2(r, L)$ and hence $\mathcal{R}(\mathcal{E}_2)[r] = v$ conforms with a location consistent execution of program $\mathcal{P}_2$.

As a concluding note, we observe that a similar monotonicity result can also be established when $\mathcal{P}_2$ is obtained by partitioning a sequential thread from $\mathcal{P}_1$ into two parallel threads (assuming that this partitioning satisfies all control and data dependences). This is another way in which $\mathcal{P}_2$ can be strictly "more parallel" than $\mathcal{P}_1$.

To the best of our knowledge, the monotonicity property holds for all memory consistency models proposed in the literature. Proving monotonicity for the LC model provides extra evidence that LC is a reasonable memory model.

3.4 Non-intrusive Reads

In this section, we show that reads are *non-intrusive* in the LC model. Consider a location consistent execution $\mathcal{E}_1$ of a given parallel program $\mathcal{P}_1$. Now consider parallel program $\mathcal{P}_2$ obtained by adding a single read operation to $\mathcal{P}_1$. We derive an execution $\mathcal{E}_2$ for program $\mathcal{P}_2$ from execution $\mathcal{E}_1$ by inserting all dynamic instances of the read operation that was added to $\mathcal{P}_1$ so as to obtain $\mathcal{P}_2$. The non-intrusive reads property guarantees that $\mathcal{E}_2$ will be a location consistent execution of program $\mathcal{P}_2$.

The proof of the non-intrusive reads property for the LC model follows simply from the fact that read operations do not appear in the pomset state of a memory location. Let $\mathcal{R}(\mathcal{E}_1)$ and $\mathcal{R}(\mathcal{E}_2)$ be the read mappings for program executions $\mathcal{E}_1$ and $\mathcal{E}_2$. Consider a read operation r on location L in execution $\mathcal{E}_1$; r must also be present in execution $\mathcal{E}_2$. Let $V_1(r, L)$ and $V_2(r, L)$ be the value sets for read operation r in executions $\mathcal{E}_1$ and $\mathcal{E}_2$ respectively. Since read operations do not appear in *state*(L), it must be the case that $V_1(r, L) = V_2(r, L)$. Note that any value $v = \mathcal{R}(\mathcal{E}_1)[r]$ returned by read operation r in execution $\mathcal{E}_1$ must satisfy $v \in V_1(r, L)$. Therefore, $v \in V_2(r, L)$ and hence $\mathcal{R}(\mathcal{E}_2)[r] = v$ conforms with a location consistent execution of program $\mathcal{P}_2$.

To the best of our knowledge, the non-intrusive reads property holds for all memory consistency models proposed in the literature. Proving the non-intrusive reads property for the LC model provides extra evidence that LC is a reasonable memory model.

4 The LC Cache Consistency Protocol

In cache-based shared-memory multiprocessor systems, the management of the cache is a vital issue that has a significant impact on system performance. The presence of copies of the same location in multiple caches requires that these copies be managed in a way that does not violate the requirements of the underlying memory consistency model. This is known as the *cache consistency* problem. Aggressive cache management schemes can exploit loose constraints in weaker memory models by reducing consistency-related traffic in the memory system.

In this section, we introduce the LC protocol[3], a new cache consistency protocol that supports the semantics of the LC model. As we will see, the partial order semantics of the LC model enables the LC protocol to be simpler and more scalable than existing protocols that must obey the memory coherence assumption.

[3] An earlier protocol in a pilot study conducted by S. Merali and G.R. Gao [19] was also called the LC protocol but that protocol maintained directories as in a directory-based protocol.

4.1 Classification of Existing Cache Protocols: Snooping or Directory-Based

Existing cache-consistency protocols for SC-derived memory models can be divided in two classes [13]:

- **Snooping**: Each cache block contains the shared status of the corresponding memory block, but no central state information is kept in memory. Snooping protocols rely on the presence of on a global shared-memory bus and on a cache controller in each processor that "snoops" on the bus transactions. If a processor contains a cached copy of the block involved in a bus transaction, the cache controller for the processor performs the appropriate action based on the state information of the cached block.

- **Directory-based**: The state of a memory block and its cached copies is maintained in a *directory*. The directory keeps track of the state changes of each cache block and takes appropriate actions to maintain coherence by sending messages to individual processors listed in the directory.

We first discuss the issues of *write-through* vs. *write-back* policies and *invalidation* vs. *write broadcast* approaches in cache consistency protocols. Maintaining the "memory coherence" requirement of the SC-derived memory consistency models is simpler when a write-through policy is assumed. However, most modern processors find the extra memory traffic caused by a write-through policy to be prohibitively high, and thus employ a write-back policy instead. In the presence of a write-back policy, both snooping and directory-based protocols maintain coherence by ensuring that at any time there is a unique "owner" of a cached block.

There are two approaches to maintaining this ownership. The first is to ensure exclusive access to a data item before a write to that item takes place — this is called the invalidation approach because other copies of the data item are invalidated before the write. The second is the write broadcast approach in which a write operation on one processor triggers a broadcast of the new value to all other cached copies. Among these two approaches, there seems to be a marked preference for the invalidation approach. Therefore, we will limit our attention in this paper to cache consistency protocols that assume a *write-back* policy and that follow an *invalidation* approach since these are the predominant assumptions in modern cache consistency protocols.

To ensure the single-ownership property, snooping protocols use the serialization of accesses on the bus as a single point of arbitration, which dictates a serialization of writes. That is, when two writes are in a race, the write that wins the arbitration will be initiated and will not complete until all its invalidation requests are satisfied. Snooping protocols have been used successfully in small-scale bus-based shared-memory multiprocessors. On the other hand, directory-based protocols have been proposed for large-scale shared memory multiprocessors that use an interconnection network instead of a bus. There is no longer a single point of arbitration. However, a single place — the directory — is provided to maintain the state

Processor 1	Processor 2		Processor N
`while(...) {`	`while(...) {`		`while(...) {`
`  . . .`	`  . . .`		`  . . .`
`  acquire(X);`	`  acquire(X);`	···	`  acquire(X);`
`  X = F_1(X);`	`  X = F_2(X);`		`  X = F_N(X);`
`  release(X);`	`  release(X);`		`  release(X);`
`  . . .`	`  . . .`		`  . . .`
`}`	`}`		`}`

Figure 3: Example of a parallel program with no data races.

information of cached blocks for each memory block. In this case a more sophisticated protocol engine is needed to ensure that unique ownership is enforced. The directory serves as a single rendezvous point for the serialization of writes.

Let us use the parallel program outlined in Figure 3 as a simple example to discuss existing snoopy and directory-based protocols. In this example, each processor executes a `while` loop; the only synchronization among the processors is through the `acquire` and `release` operations on the shared variable X. Each processor uses the acquire-release construct to perform some (processor-specific) read-modify-write sequence on X in mutual exclusion. The acquire-release operations make the execution of this parallel program nondeterministic, but all executions of the program are assumed to be free of data races. Self-scheduled execution of a parallel loop [22] is a simple example that fits the parallel program structure shown in figure 3.

Consider the execution of the program in figure 3 with a snooping protocol. Executing a write operation of X in the acquire-release construct will force an invalidation of cached copies of X on all other processors, so as to change the ownership from "shared" to "exclusive". This invalidation is usually accomplished by broadcasting an invalidate message on the bus. Though the broadcast is convenient to perform on a bus, it still consumes an entire bus cycle and leads to extra bus traffic.

Directory-based protocols avoid the broadcast overhead incurred for write operations in snooping protocols by instead having the owner send invalidate messages to only those processors listed in the directory as containing cached copies of the block being written into. This can lead to a savings if only a small number of processors contain cached copies. However, even in that case, the savings comes at the cost of the extra complexity of maintaining directories in the cache consistency protocol.

4.2 A New Cache Protocol Based on the LC Model

Under the LC model the only requirement of a memory system is to obey the ordering dictated by the partial order semantics, which is specified by the programming model and memory abstractions (see Section 2). Since there are no memory-coherence or serialization requirements beyond what is implied by the program's partial order, a cache consistency protocol for the LC model does not need to ensure single ownership of memory blocks. Therefore, we can develop new cache protocols to support the LC model. The properties that we desire for such new protocols are that they:

- should not be directory-based, thus avoiding the cost associated with the maintenance of a directory,
- should not rely on snooping or on any centralized arbitration to maintain coherence, thus eliminating a synchronization bottleneck and allowing for the construction of scalable machines,
- and should avoid invalidate requests for maintenance of memory and cache coherence.

Intuitively, since no memory coherence requirement is imposed, these new cache protocols should guarantee that a read operation always returns an element of the "value set" specified by the LC model (see section 2.2). The solution adopted in this paper is to always keep a valid value in the main memory location. Thus, any read miss in a cache will always find a legal value in the corresponding memory location.

Our new cache protocol only needs to update the main memory location. Since the value set of a read operation can contain multiple legal values, there is no need to invalidate other cached copies of the memory. In other words, memory and caches do not need to be coherent! Several caches might contain different (but legal) values of the same memory location. Consequently our new protocol does not require a directory or the use of a bus snooping mechanism to maintain cache consistency.

An outline for the LC cache consistency protocol is as follows. In order to simplify the initial discussion, we assume that each cache line contains a single word (location). An extension to a cache with multi-word lines is presented in section 4.4. The LC protocol assumes that each cache line may be in one of the following states:

invalid — the cache line does not contain valid information. A read or write operation to the line will result in a *miss*.

clean — the cache line contains valid information. A read or write operation to the line will result in a *hit*. If the cache line receives a "self-invalidation" it will go to an *invalid* state. A self-invalidation might result from an **acquire** operation performed in the same processor as explained later.

dirty — the cache line contains valid information. A read or write operation to the line will result in a *hit*. The cache line will remain in the dirty state even if it is the target of a self-invalidation operation.

The above three states can be easily implemented through the use of two state bits: a *valid bit* and a *dirty bit*. Table 1 illustrates the correspondence between the three cache states and the two state bits.

valid bit	***dirty bit***	**State**
0	0	*invalid*
1	0	*clean*
1	1	*dirty*
0	1	Not a legal state

Table 1: Correspondence between cache states and state bits.

The state information for the LC protocol is only maintained in each processor's cache; no form of directory information or state information needs to be maintained in the memory. We observe that the two state bits per cache line required by the LC protocol are already present in many uniprocessor cache implementations in modern microprocessors. The main difference from the uniprocessor case is in the use of these state bits by the LC cache protocol to determine what consistency actions should be performed for *acquire* and *release* synchronization operations.

Specifically, the actions performed by the LC protocol for read/write/release/acquire operations are as follows:

Read Operation

- If the line X is in a *clean* or *dirty* state, read(X) results in a **hit**. The value in the cache is a legal value to be returned by the operation read(X).
- If the line X is in an *invalid* state, read(X) results in a **miss**. The line X of the cache will be fetched from the memory and stored in the cache in a clean state. Our protocol guarantees that the value brought from the memory is a legal value.

Write Operation

- If the line X is in a *dirty* state, operation write(X) will simply write the new value in the line.
- If the line X is in a *clean* or an *invalid* state, operation write(X) will write the new value in the line and change the line into a *dirty* state.

Acquire Operation

When executing an acquire(X) operation a processor p first performs the atomic hardware operation used to get the lock, and then performs the following consistency operations:

- If the cache line X is in a *clean* state, p will invalidate X in the cache. This *self invalidation* by processor p is necessary because the last processor to perform a release(X) may have written a value for X that p must observe in accordance with the LC model.
- If the cache line X is in a *dirty* or *invalid* state no state change is necessary.

Release Operation

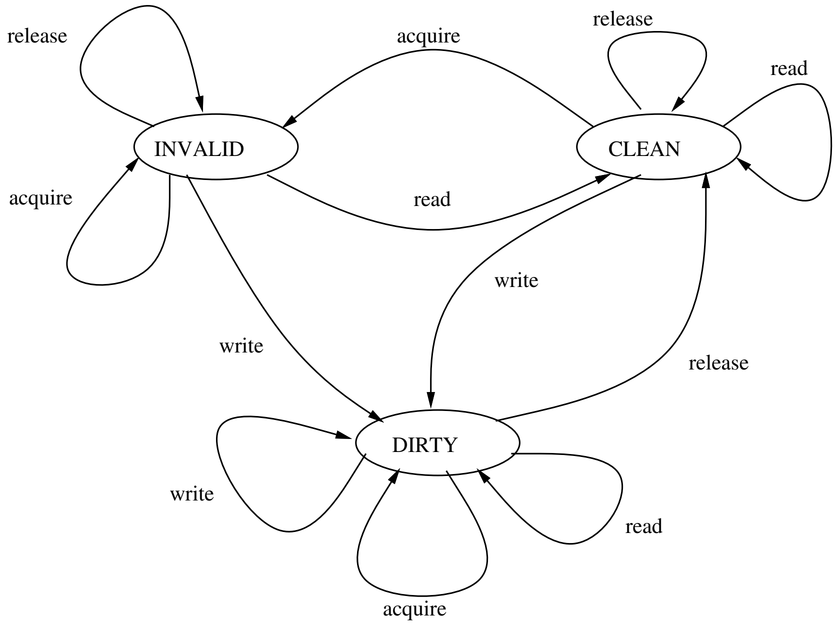

Figure 4: State transition diagram for LC protocol

- If the line X is in a *dirty* state, operation release(X) will update the memory, wait for an acknowledgment, and set the cache line X to a *clean* state.
- If the line X is in a *clean* or an *invalid* state, operation release(X) will not incur in any state change.

To complete the release operation, the processor also needs to give up the lock after performing the above consistency actions. We assume that all write operations to X in the same processor must complete before the release(X) operation can complete. Thus, if a write buffer is used, a release(X) operation cannot complete till all write operations to X have been transfered from the write buffer to memory. The processor then waits for an acknowledgment from memory that all its write operations to X have completed. Note that this wait for write completion does not require exclusive ownership.

Under the LC protocol, a write operation is always locally performed in the cache. No consistency-related actions, such as obtaining exclusive ownership of the location before the write, are necessary. When a cache line X that is in a *dirty* state needs to be ejected to make room for a new value brought in by a read miss the value of X is written back to memory, but no other actions are needed.

Figure 4 presents the state transition diagram for the proposed LC cache protocol. Note that this state diagram is much simpler than the state diagram for traditional cache coherence protocols such as those described in [14]. For example, the LC protocol does not need to send any global invalidation messages to other caches. All the operations presented in the state transition diagram are performed by the local processor (e.g. read, write, release, acquire).

For the sake of comparison, consider the execution of the data-race-free example program in figure 3 with the LC protocol. Each processor performs a "self-invalidate" of X when it acquires the lock for X, thus ensuring that receives the most recent value from memory. In addition, each processor writes back its most recent value of X to memory when performing a release operation. There is no broadcast of invalidates as in a snooping protocol, or sends of multiple invalidate messages to multiple readers as in a directory-based protocol.

We conclude this section with a theorem that establishes the correctness of the LC cache consistency protocol.

Theorem 4.1. *The LC cache protocol obeys the LC memory consistency model.*

Proof: We have to show that the value returned by any read operation belongs to the set of values in the corresponding value set. Without loss of generality, consider a read operation r in processor P_j. There are three cases:

i. **The read is a hit in a dirty line.** The line must have been written by an early write from the same processor P_j and there was no release operation after the write. If there had been a release operation, it would have found this line dirty and would have updated the memory and changed its state to clean. Therefore the value stored in the cache line is a legal value in the corresponding value set.

ii. **The read is a miss.** In this case, the value v returned by the read miss is delivered from the memory location. Assume that this value was written by a write operation $w = write(P_k, v)$ performed by processor P_k. There are three possible cases for how value v written by write operation w in processor P_k might have reached the memory location:

    (a) Processor P_k had the location cached in a *dirty* state and then performed a release operation on the location causing value v to be written back to memory by the LC protocol.

    (b) Processor P_k had the location cached in a *dirty* state and incurred a conflict/capacity miss that caused value v to be written back to memory by the LC protocol.

    (c) Write operation w is the initial write of undefined value $\perp$. (Recall from section 2.2 that we assume that each memory location is initialized with a write of an undefined value at the start of program execution.)

    In the first two cases, the value was copied to the memory from a dirty cache location. We will show (by contradiction) that no other write operation $w' = write(P_h, v')$ could have been performed on the location such that $w \prec w' \prec r$ and $v' \neq v$. Assume that there exists such a write operation, w'. We have to consider two subcases:

- w' was issued on the same processor $P_h = P_k$ that wrote value v. In this case the value present in the cache location, and thus written back to memory, must be v' and not v. Contradiction.
- w' was issued in a processor $P_h \neq P_k$. In this case we have that $processorset(w) \cap processorset(w') = \emptyset$ and therefore $w \not\prec w'$. Contradiction.

In the third case above, the value returned by read operation r is the initial undefined value, $v = \perp$, for the location created by (pseudo) write operation w. Again, we will show (by contradiction) that no other write operation $w' = write(P_h, v')$ could have been performed on the location such that $w \prec w' \prec r$ and $v' \neq \perp$. Assume that there exists such a write operation, w'. There are two subcases as before:

- w' was issued on the same processor $P_h = P_i$ that read value v. In this case, the value present in the memory location can no longer be $\perp$ since the location must at least have received value v' from write w'. Hence the value returned by read operation r is not $\perp$. Contradiction.
- w' was issued in a processor $P_h \neq P_i$. In this case we have that $processorset(w') \cap processorset(r) = \emptyset$ and therefore $w' \not\prec r$. Contradiction.

This concludes the proof for the case when read operation r is a miss.

iii. **The read is a hit in a clean line.** If the cache line is in a clean state, we have to consider two situations:

(a) **The cache location was updated by a prior cache miss on processor P_j, but the memory location was since changed by a write back or release operation on processor $P_k \neq P_j$.** If the cache in processor P_j is still in a clean state it means that P_j has not issued an acquire operation for this location — meaning that the programmer did not intend for this value to be synchronized. Therefore the value stored in the clean line of processor P_j is a member of the value set for the read operation, and hence still a legal value according to the LC model.

(b) **The cache location has the same value as the corresponding memory location.** This is similar to the read miss case discussed above where the value returned by the read operation is the same as the value in the corresponding memory location.

Having considered all possible cases for a read operation, we conclude our proof that the value returned by any read operation is a legal value in the corresponding LC value set. □

4.3 Comparison with Existing Cache Protocols

In this section, we provide a qualitative comparison of the LC protocol with existing snooping and directory-based cache consistency protocols. The main advantages of the LC protocol are its *simplicity* and its *scalability*. The simplicity comes from the fact that the LC protocol does not require extra hardware either to snoop on a bus or to maintain a directory. The scalability comes from the fact that the LC protocol does not perform invalidations on multiple processors when a variable is acquired or written. In contrast, a snooping protocol has to broadcast invalidates to all processors when performing an acquire or a write operation. A directory-based protocol has to send invalidate messages to all processors that have a cached copy of the location being written or acquired.

The LC protocol derives these benefits from the underlying LC model. The fact that the LC model is weaker than RC and other consistency models based on memory coherence enables the LC protocol to be simpler than consistency protocols for memory models based on the memory coherence assumption. If either the snooping protocols or the directory-based protocols were relaxed so as to only perform self-invalidates as in the LC protocol, they would no longer be correct protocols for their underlying memory consistency model.

The properties described in section 3 have already established the usefulness of the LC model. In particular, the Equivalence Property guarantees that the LC model is equivalent to the RC model for parallel programs that have no data races. For such programs the simpler and more scalable LC protocol is guaranteed to implement the same memory consistency semantics as the snooping and directory-based protocols for the SC and RC models. We saw an illustration of this in section 4.2 in the discussion of how the LC protocol operates on the parallel program in Figure 3.

For programs with data races, it is the job of the memory consistency protocol to implement the semantics of data races specified by the underlying memory model. We advocate the use of the LC model because it guarantees important properties such as the Equivalence Property, while being strictly weaker than other memory consistency models that guarantee the same properties. In snooping and directory-based protocols the extra overhead of supporting memory coherence in programs with data races is incurred by all programs, including programs that are free of data races. It is not easy to disable this support because it is not possible to determine at the start of a program's execution whether or not the execution will exhibit a data race. This is unfortunate because it is desirable for parallel programs be data-race-free, and in fact most parallel program executions in practice do not exhibit data races.

The LC model ensures that data-race-free programs are not penalized by extra consistency overhead that would only be necessary for programs with data races. Instead, the LC model provides a weaker semantics for programs with data races; if there is a need to write a parallel program using the LC model that has data races and that satisfies the memory coherence assumption for some variable(s), each memory access to the variable(s) can be enclosed in an

Processor 1	**Processor 2**		**Processor N**

```
while(...) {
  . . .
  acquire(X);
  X = F_1(X);
  release(X);
  . . .
  read X;
  . . .
}
```

```
while(...) {
  . . .
  acquire(X);
  X = F_2(X);
  release(X);
  . . .
  read X;
  . . .
}
```

...

```
while(...) {
  . . .
  acquire(X);
  X = F_N(X);
  release(X);
  . . .
  read X;
  . . .
}
```

Figure 5: Example of a parallel program with data races.

acquire-release construct[4]. Thus, in the LC model and the LC protocol, the extra overhead of maintaining memory coherence is only incurred by programs that have data races and that need the memory coherence property.

To demonstrate how the LC protocol works for a program with data races, consider the relaxation-style parallel program outlined in Figure 5. The main difference with the program in Figure 3 is that each processor now performs an asynchronous read of X in each iteration of the `while` loop. In this example, we assume that there are no asynchronous write operations. The presence of asynchronous reads is sufficient to cause data races because there may be multiple candidate write operations that can supply the value of the read operation. However, the uniprocessor ordering of the *read X* operation following the *release(X)* ensures that the result of the read must either come from the most recent write performed in the same processor or from a *later* write operation on another processor. This kind of guarantee can be sufficient to ensure termination in many relaxation algorithms.

With the LC protocol, the acquire-release construct will execute just like the execution of the acquire-release construct in figure 3 discussed in section 4.2. For the asynchronous read, there are two cases:

Case 1: X remained in the processor's cache between *release(X)* and *read X*
In this case, the LC protocol will just return the value of X in cache which will still be valid.

Case 2: X was evicted from the processor's cache between *release(X)* and *read X*
In this case, the read operation will incur a cache miss just as in the uniprocessor case. The LC protocol would have ensured that the value in memory was written back either by the most recent *release(X)* on the same processor or by a later *release(X)* performed on another processor.

Therefore, each acquire/release/read/write operation incurs a constant time overhead without involving additional processors.

[4]The insertion of acquire-release operations for memory coherence can be done manually by the programmer or automatically by the computer system at compile-time or run-time.

With a snooping or directory-based protocol, each write operation will cause all N copies of X to be invalidated on N processors. Therefore, $O(N)$ overhead is incurred for each write operation in these protocols, which is an order-of-magnitude larger than the constant-time overhead incurred in the LC protocol. This example demonstrates the scalability of the LC protocol compared to snooping or directory-based protocols.

4.4 Extensions to the LC Protocol

We describe a straightforward extension of the LC protocol to caches with multi-word (multi-location) lines. The basic idea is as follows: the state information described in section 4.2 is maintained for each location in a line. The main effect of having multiple locations per cache block can be thought of as implementing anticipatory prefetches of independent locations to exploit spatial locality.

As a result, different locations in a cache line can be in different states! When a cache line is brought into the cache, all invalid locations in the line are fetched into the cache in clean state. When a cache line is ejected from cache, the consistency actions are performed for each location that is being ejected. In other words, the ejection of a line from the cache can be thought of as a *block ejection* of multiple independent locations.

We illustrate the general mechanism by describing how *read* and *write* operations would work in the extended setting. The *release* and *acquire* operations can be extended in a similar fashion:

- **Read operations**

  A *read* to a location L **hits** in a cache if the line containing the location is in the cache and the location within the line contains valid data (i.e if the location is *not* in *invalid* state). In this case the value of location L in the cache line is a legal value to be returned by the *read*. Otherwise, the *read* is said to **miss**. All read-misses in a cache are guaranteed to be legally serviced by memory. Our LC-based protocol guarantees that if a *read* access cannot be serviced by a cache, the value retrieved from memory is a legal value.

  A *read* to a location L **misses** if the corresponding line is not in the cache, or if the location within the line is in *invalid* state. When a read-miss occurs, the location is fetched from memory and put in *clean* state. All other locations that are fetched from memory are also put in the *clean* state. We assume that the cache replacement mechanism will take care of writing back locations in the replaced line that are in *dirty* state.

- **Write Operations**

  When a *write* operation to location L is executed, it will check the cache state of the location L. If L is in a *dirty* state, it will simply write the value into the location of the corresponding cache line. Otherwise, if it is in *clean* state, then it writes the value in the location of the line and change the state into *dirty* state. In case it is in *invalid* state (i.e. result in a write cache miss), then it writes the value in the location of the line and change the state into dirty state. If the line containing location L is not in the cache, a separate replacement mechanism will make room in the cache for the new line, location L will be written and put in *dirty* state and all the other locations will be in *invalid* state[5].

Even in the case of multiple locations per cache line, all consistency state information is maintained at the location level in this extension. This extension implies that two state bits are needed for each word in the cache, which leads to extra cost in the cache support of some modern processor architecture. For example, the Pentium-Pro and the MIPS-4000 processors maintain only one valid and one dirty bit per cache line. The cost of the extension is lowered if the processor maintains a valid bit and a dirty bit per subblock, as in the Alpha 21164 processor. Finally, if it is not possible to maintain valid and dirty bits on a location level, our plan is to extend the LC protocol along the lines of the data merging protocol reported in [15].

4.5 Optimization of the LC Cache Protocol

So far, our discussion on the LC cache protocol is based directly on memory operations such as read, write, acquire, and release. However, further refinement and optimization is possible and desirable.

For example, examine the code section (a) shown in Figure 6. This is a critical-section, where the variable X is read twice and updated once. In code section (b), we show how the code is refined. We separate the two functions of the acquire/release operations — the lock/unlock operations and consistency related operations. That is, acquire and release operations are replaced by normal lock/unlock operations which are commonly supported as low level primitives in modern processor architectures. The consistency related operation associated with acquire and release are now implemented by "refresh" and "write-back" operations.

We introduce a "refresh" operator before each read operation. A *refresh(X)* operation will perform a self-invalidation of X following the same rule used for an acquire operation. We also introduce a "write-back" operation after each write operation. The "write-back" operation plays the role of a synchronization with the final "sync-writeback" operation to be placed before the corresponding release operation. The sync-writeback operation can only complete when it has received all synchronization signals from the write-back operations in the critical section. This ensures that the unlock is not performed until all the write operations to X have been completed.

The refinement presented in this section can be an optimization if, for example, the read operations are conditionally executed. In this case, a *refresh(X)* operation is performed only

[5] Notice that this assumes word placement in the cache. It is a simple matter to extend the protocol to allow line or subblock placement.

before refinement	after refinement
`acquire(x);`	`lock;`
`. . .`	`. . .`
	`refresh(X);`
`read X;`	`load(X);`
`. . .`	`. . .`
	`refresh(X);`
`read X;`	`load(X);`
`. . .`	`. . .`
`write X;`	`store X;`
	`write_back X;`
`. . .`	`. . .`
`. . .`	`sync-writeback`
`release X;`	`unlock`
(a)	(b)

Figure 6: Example of a refinement and optimization

when a read(X) operation is performed; the refresh(X) operation need not be performed at each acquire(X) operation as described in section 4.2. Therefore, this refinement can be useful in removing some unnecessary self-invalidation operations at runtime. We anticipate several opportunities for compiler optimization after this kind of refinement is performed, such as global code motion to minimize the overhead of consistency related operations.

5 Related Work

Related work on cache-coherence protocols has been studied widely and the readers should find an excellent introduction in [14] and other references therein. We have also given a brief review of issues related to this paper in Section 4. We focus our discussion of related work in this section on *relaxed* memory models which attempt to relax the restrictions on memory access ordering imposed by the sequential consistency (SC) model so as to enhance performance.

In order to alleviate the SC performance limitations, designers have proposed models that guarantee the SC interface for a restricted set of programs, but allow optimizations to be applied safely. The looser models share the common intuition that if a program has enough synchronization then the program can appear to execute in a sequentially consistent manner without hardware support for sequential consistency. In other words, the results of every run of a "properly synchronized" program are guaranteed to be consistent with some sequentially consistent run of the program. We present these relaxed consistency models in the chronological order in which they appeared in the literature: Release Consistency (1990) in section 5.1, Lazy Release Consistency (1992) in section 5.2, Entry Consistency (1993) in section 5.3, and Dag

Consistency (1996) in section 5.4. (Our past work on Location Consistency [9, 10] appeared in the 1994-1995 time frame.)

5.1 Release Consistency

The goal of Release Consistency (RC) is "to exploit additional information about shared accesses to develop a memory consistency model that allows for more efficient implementations" [12]. The RC model distinguishes between ordinary shared memory accesses and synchronization instructions, and further distinguishes between *acquire* and *release* synchronization operations.

For example, in the RC protocol, a *release* triggers a write of a shared variable and an *acquire* triggers the read of a shared variable. The purpose of a *release* is to inform other processes that all accesses that precede it (in program order) have completed. Similarly, the purpose of an *acquire* is to await such a signal from another processor before initiating any further accesses.

RC guarantees sequential consistency for a specific class of programs which it classifies as "properly labeled". Intuitively, a program is properly labeled if there is enough synchronization so that for all legal interleaving of accesses, pairs of conflicting ordinary accesses are separated by a *release-acquire* chain i.e., the program has no data races.

The RC condition has imposed certain limitations on how memory operations should be ordered. In the DASH implementation of RC [18], each write operation must be explicitly acknowledged to ensure atomicity and thus satisfy the memory coherence condition. In addition, the release operation contains an implicit "fence" operation which ensures that all previous memory operations must be completed before subsequent operations can be issued.

5.2 Lazy Release Consistency

Lazy Release Consistency (LRC) can be viewed as an extension to RC aimed at reducing the number of messages and the amount of data exchanged in a distributed shared-memory system implemented in software [16]. The basic motivation for the algorithm is the observation that cross-processor consistency information only needs to be propagated at acquire synchronization points, at which point RC requires *all ordinary accesses that precede the corresponding* release to be performed with respect to the acquiring processor. While the *eager* counterparts of LRC[6] make modifications globally visible at the time of a *release*, LRC exploits the intuition that only the processor that *acquire*s a variable needs to see all modifications that *precede* the *acquire*. So, whereas in an eager implementation, consistency broadcasts are made at *release* points and may involve all processors, in the lazy implementation, the broadcasts are made at the points of *acquire*, and involve only the acquiring and releasing processors.

[6]For instance, the RC implementation on Munin [5].

In the DSM implementation of LRC [16], coherence actions only take place at the acquire points, while release operations involve no coherence actions. It is found that both the number of messages as well as the amount of data exchanged are generally smaller for LRC than an eager implementation of RC.

5.3 Entry Consistency and the Midway System

Entry Consistency (EC) [2] can also be viewed as a further relaxed extension of RC. The basic difference is the following: whereas in RC, a synchronization object (variable) protects access to all shared data, in EC, there is an explicit correspondence between synchronization variables and the shared data they guard. As in LRC, modifications are propagated at an acquiring synchronization, but now, only the shared data that the synchronization variable guards is guaranteed to be consistent at that point. This correspondence between shared and synchronization data, which are implicit in the structure of a parallel program, is required by EC to be made explicit to the compiler and run-time system. An aggressive implementation can make use of this information to reduce the number of consistency messages (e.g. cache invalidations and/or updates) flowing across the system.

As in previous models, programs that include all the necessary labeling information, and have no data races (i.e., are "properly synchronized") observe a sequentially consistent shared memory. Measurements made on the implementation of EC on Midway [2] show that a program written for EC requires substantially fewer consistency transactions than stronger models such as RC. However, the caching protocol is still essentially based on a "single-ownership" model.

5.4 Dag Consistency

Blumofe et al [3] defined the *dag consistency* memory model for deterministic spawn-sync multithreaded programs in which dynamic program execution is modeled as a computation dag (directed acyclic graph) of non-suspensive "threads". Threads are created as follows. Main program execution begins in a single thread. A spawn (fork) statement creates a procedure call in a new thread that can execute concurrently with the caller. A sync (join) statement terminates the executing thread and creates a new thread for the computation that follows the sync statement. This new thread must wait for all threads spawned by the previous thread to terminate before it can start execution. Each vertex in the computation dag corresponds to a distinct thread. An edge in the computation dag represents a partial order constraint *i.e.,* an edge exists between the caller thread and the callee thread in a spawn statement and between each previously spawned thread and the continuation thread in a sync statement. Thus, the computation dag defines a partial order on threads. The computation dag is an abstraction used in defining dag consistency, and is not actually computed at runtime.

The shared memory of a multithreaded computation is said to be dag-consistent if the following two conditions hold:

1. When thread i reads a memory location, it receives a value that was written by some thread j such that $i \not\prec j$ in the computation dag.
2. For any three threads i, j, k such that $i \prec j \prec k$, if k reads a location written by both i and j then the value read by thread k is not the one written by thread i.

These conditions for dag consistency appear to be identical to the conditions for location consistency defined in [9, 10] when applied to the special case of deterministic spawn-sync programs considered in [3]. For deterministic programs, the set of most recent writes for a location in the LC model will always either be an empty set or a singleton set.

6 Conclusions

In this paper, we argued that the past trend of including the memory coherence assumption in all memory consistency models should be re-examined. We believe that this assumption imposes serious limitations in the design of scalable shared-memory multiprocessors. We propose that memory consistency models should instead be based on the partial order execution semantics of parallel programs without relying on the memory coherence assumption.

We defined a new memory model called Location Consistency (LC) in which the state of a memory location is modeled as a partially ordered multiset (pomset) of write operations and synchronization operations. We established the usefulness and robustness of the LC model by proving the weakness, equivalence, monotonicity and non-intrusive properties for the LC model. We introduced a new cache consistency protocol for the LC model. This LC protocol is simple and scalable. It is unique in its support of the LC model without incurring any overhead of bus snooping or of maintaining directories.

Acknowledgments

This research is supported in part by a National Science Foundation (NSF) grant MIPS-9707125, and a DARPA and NSA grant MOD706097. The authors would like to thank Professor Arvind and Professor Leiserson at M.I.T. and their students for their feedback on earlier discussions of the LC model. The authors are also grateful to Dr. José Nelson Amaral for his help in proofreading and editing the manuscript, for his many suggestions that improved the paper, and for his contributions to the proof of Theorem 4.1.

References

[1] 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.

[2] B. Bershad, M. Zekauskas, and W. Sawdon. The midway distributed shared memory system. In *Proceedings of the IEEE COMPCON*, 1993.

[3] Robert D. Blumofe, Matteo Frigo, Christopher F. Joerg, Charles E. Leiserson, and Keith H. Randall. An analysis of dag-consistent distributed shared-memory algorithms. In *Proceedings of the 8th Annual ACM Symposium on Parallel Algorithms and Architectures*, pages 297–308, Padua, Italy, June 24–26, 1996. SIGACT/SIGARCH.

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

[5] John B. Carter, John K. Bennett, and Willy Zwaenepoel. Implementation and performance of Munin. In *Proceedings of the Thirteenth ACM Symposium on Operating Systems Principles*, pages 152–164, Pacific Grove, California, October 13–16, 1991. ACM SIGOPS. *Operating Systems Review,* 25(5), 1991.

[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] 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.

[8] Matteo Frigo. The weakest reasonable memory model. Master's thesis, Massachusetts Institute of Technology, 1998.

[9] Guang R. Gao and Vivek Sarkar. Location consistency: Stepping beyond the barriers of memory coherence and serializability. ACAPS Technical Memo 78, School of Computer Science, McGill University, Montréal, Québec, December 1994. In ftp://ftp-acaps.cs.mcgill.ca/pub/doc/memos.

[10] Guang R. Gao and Vivek Sarkar. Location consistency: Stepping beyond memory coherence barrier. volume II, pages 73–76, 1995.

[11] Guang R. Gao and Vivek Sarkar. On the importance of an end-to-end view of memory consistency in future computer systems. In *Proceedings of the International Symposium on High Performance Computing*, pages 30–41, Fukuoka, Japan, 1997.

[12] 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.

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

[14] John L. Hennessy and David A. Patterson. *Computer Architecture: A Quantitative Approach*. Morgan Kaufmann Publishers, Inc., San Francisco, California, 2nd edition, 1996.

[15] Alan H. Karp and Vivek Sarkar. Data Merging for Shared-Memory Multiprocessors. *Proceedings of the 26th Hawaii International Conference on System Sciences, Wailea, Hawaii, Volume I (Architecture)*, pages 244–256, January 1993.

[16] 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.

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

[18] Daniel Lenoski, Kourosh Gharachorloo, James Laudon, Anoop Gupta, John Hennessy, Mark Horowitz, and Monica Lam. Design of scalable shared-memory multiprocessors: The DASH approach. In *Digest of Papers, 35th IEEE Computer Society International Conference, COMPCON Spring '90*, pages 62–67, San Francisco, California, February 26–March 2, 1990. IEEE Computer Society Press.

[19] Shamir Merali. Designing and implementing memory consistency models for shared-memory multiprocessors. Master's thesis, McGill University, Montréal, Québec, April 1996.

[20] 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.

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

[22] Peiyi Tang and Pen-Chung Yew. Processor Self-Scheduling for Multiple-Nested Parallel Loops. *Proc. of the 1986 Int'l Conf. on Parallel Processing*, February 1986.