An Abstract State Machine Specification and Verification of the Location Consistency Memory Model and Cache Protocol

Charles Wallace
(Computer Science Dept.,
Michigan Technological University, Houghton, MI, USA
wallace@mtu.edu)

Guy Tremblay
(Dépt. d’informatique,
Université du Québec à Montréal, Montréal, QC, Canada
tremblay@info.uqam.ca)

José N. Amaral
(Computing Science Dept.,
University of Alberta, Edmonton, AB, Canada
amaral@cs.ualberta.ca)

Abstract: We use the Abstract State Machine methodology to give formal operational semantics for the Location Consistency memory model and cache protocol. With these formal models, we prove that the cache protocol satisfies the memory model, but in a way that is strictly stronger than necessary, disallowing certain behavior allowed by the memory model.

Key words: Requirements/Specifications, Multiprocessors, Shared Memory, Cache Memories

Categories: B.3.2, C.1.2, D.2.1

1 Introduction

A shared memory multiprocessor machine is characterized by a collection of processors that exchange information with one another through a global address space [Adve and Gharachorloo 95, Culler et al. 99]. In such a machine, processors access memory locations concurrently through standard read and write instructions. Shared memory machines have various buffers where data written by a processor can be stored before it is shared with other processors. Thus, multiple values written to a single memory location may coexist in the system. For instance, the caches of various processors might contain different values written to the same location.

The programs running on a shared memory machine are affected by the order in which memory operations are made visible to processors (which previous write operations are currently visible). A memory consistency model is a contract between a program and the underlying machine architecture that constrains the order in which memory operations appear to be performed with respect to one another (i.e., become visible to processors) [Culler et al. 99]. By constraining the order of operations, a memory consistency model determines which values can legally be returned by each read operation. The implementation of a memory
consistency model in a shared memory machine with caches requires a cache protocol, that invalidates or updates cached values when they no longer represent legal readable values.

The most common memory consistency model, **sequential consistency (SC)** [Lamport 79], ensures that memory operations performed by the various processors are serialized (i.e., seen in the same order by all processors). This results in a model similar to the familiar uniprocessor model. A simple way to implement SC on a shared memory multiprocessor is to define a notion of ownership of a memory location and to require a processor to become the owner of a location before it can update its value. The serialization of memory operations is obtained by restricting ownership of a location to one processor at a time.

Under the SC model, there is always a unique most recent write to a location. All other values stored in the system for that location are not legally readable and must be either invalidated or updated. Thus a major drawback of SC is the high level of interprocessor communication required by the cache protocol. Because of the requirement that all write memory operations be serialized, the SC model is quite restrictive and is thus said to be a strong memory model. Weaker memory models have been proposed to relax the requirements imposed by SC. Examples include release consistency [Gharachorloo et al. 90], lazy release consistency [Keleher et al. 92], entry consistency [Bershad et al. 93], DAG consistency [Blumofe et al. 96], and commit, reconcile and fences (CRF) [Shen et al. 99]. Relaxed memory models place fewer constraints on the memory system than SC, which permits more parallelism and requires less interprocessor communication but complicates reasoning about program behavior.

All these models have the coherence property. In a coherent memory model, all writes become visible to other processors, and all the writes in the system are seen in the same order by all processors. In 1994, Gao and Sarkar proposed the **Location Consistency (LC) memory model** [Gao and Sarkar 94], one of the weakest memory models proposed to date. LC is the only model that does not ensure coherence. Under LC, memory operations performed by multiple processors need not be seen in the same order by all processors. Instead, the content of a memory location is seen as a partially ordered set of values. Because LC allows the coexistence of multiple legal values to the same location, there is no need to invalidate or update remote cached values. Hence the LC model has the potential to reduce the consistency-related traffic in the network.

In their more recent paper [Gao and Sarkar 00], Gao and Sarkar describe both the LC memory model and a cache protocol, the **LC cache protocol**, that implements the LC model. They describe the LC model in terms of an “abstract interpreter” that maintains the state of each memory location as a partially ordered set of values, thus defining the set of legal values for a read operation. The LC cache protocol is designed for a machine in which each memory location has a single value stored in main memory and each processor may cache values for multiple locations.

An important requirement of a cache protocol is that the resulting machine behavior be allowed by the corresponding memory model. Gao and Sarkar present a proof that the cache protocol satisfies the memory model. However, their description of the memory model is based on an ad hoc operational semantics notation that is not rigorously defined. The description of the cache protocol is entirely informal and leaves some important assumptions unstated.

In this paper, we specify the LC memory model and the LC cache protocol
using the ASM approach [Gurevich 95]. We use the original descriptions by Gao and Sarkar as the basis for our models, making explicit some of the assumptions present in the original descriptions. We then prove that the LC cache protocol correctly implements the LC memory model, \textit{i.e.}, for a machine that implements the cache protocol, every read operation returns a legal value according to the memory model. In addition, we show that the LC cache protocol is \textit{strictly stronger} than the LC memory model, \textit{i.e.}, there are legal values according to the memory model that cannot be returned under the cache protocol.

Our specifications of the LC memory model and of the LC cache protocol are similar in that they refine a common (top-level) specification. In [Section 2] we define the common portions of the two models, producing a model $LC_0$. In [Section 3] we refine this model to arrive at a model $LC_{mm}$ of the LC memory model. In [Section 4] we make different refinements, resulting in a model $LC_{cp}$ of the LC cache protocol. In [Section 5] we prove that $LC_{cp}$ is an implementation of $LC_{mm}$. In [Section 6], we prove that $LC_{cp}$ is in fact strictly stronger than $LC_{mm}$. Finally, in [Section 7], we present related work and we conclude in [Section 8] with directions for future work.

2 Shared Memory System and Memory Operations

In a shared memory machine, processors can reference a shared set of memory locations, organized in a global shared address space, using the same operations they use to access their local memory. Although it may appear intuitive to think that each shared memory location holds a single value at any given time, cache memories provide multiple places to store values for a single location. Thus, at any given time, multiple values may be simultaneously associated with the same memory location.

A processor can perform four types of operation on a memory location:

- A \textit{read} retrieves a value associated with a location, possibly storing it in some area local to the processor.
- A \textit{write} adds a value to the set of values associated with the location. In any real system, the number of places available to store the values associated with a location is finite. Therefore, a side effect of a read or write operation is that a value previously associated with a given location may no longer be available in the system.
- An \textit{acquire} grants exclusive ownership of a location to a processor.\footnote{In SC, only a processor that owns a location may perform a read or write on it. On the other hand, although LC has a notion of exclusive ownership, it allows processors without ownership to perform reads and writes.} The exclusive ownership of a location imposes a sequential order on processor operations. Hence when it is useful to have a unique global “most recent write” to a location, such write can be defined as the most recent write by a processor that owned the location at the time of the write. When acquiring a location, a processor updates its own state by discarding any old value it has stored for the location.
- A \textit{release} operation takes exclusive ownership away from a processor. Any processor attempting to acquire a location currently owned by another processor must wait until the location is released by its current owner. If the
releasing processor has written to the location, the release operation has the additional effect of making the value of its most recent write available to other processors. In this way, a processor that subsequently acquires the location will have access to the value of the global “most recent write”.

Gao and Sarkar do not speak of acquire and release operations separately; rather, they speak of acquire-release pairs of operations. Thus it is assumed that a processor must gain ownership of a location through an acquire before releasing that location.

The model of the LC memory model (\(LC_{mm}\)) in [Section 3] and the model of the LC cache protocol (\(LC_{cp}\)) in [Section 4] both require formalizations of the notions discussed above. In the rest of this section, we define a higher-level model \(LC_0\) to represent these notions in a generic way. \(LC_0\) models only the general control flow associated with the execution of the memory operations, including the waiting associated with an acquire operation. In this initial model, the flow of data is ignored. Later, we refine \(LC_0\) to the models \(LC_{mm}\) and \(LC_{cp}\), adding details appropriate to each of these models (partial order of operations vs. cache information).

**\(LC_0\): Universes and Agents**

In this section, we present the universes used in all our ASM models. We assume that the multiprocessor system has a fixed set of processors, a fixed set of memory locations, and a fixed set of data values. These sets are represented in \(LC_0\) as the Processor, Location and Value universes, respectively. We also define an OpType universe, comprising the four types of operation: read, write, acquire, and release.

A distributed computation in ASM is modeled as a multi-agent computation in which agents execute concurrently and where each agent performs a sequence of state transformations. In modern multiprocessors, a single processor may perform operations on different locations concurrently. Multiple processors may also perform concurrent memory operations, either on the same location or on different locations. On the other hand, a given processor cannot perform multiple concurrent memory operations on a given location. Therefore, in our abstract model, for each processor \(P\) and for each location \(\ell\) there is a unique agent whose task is to perform operations on \(\ell\) on behalf of \(P\). We call such agents processor agents and we define a universe ProcAgent accordingly. Later, we complete the definition of \(LC_0\) by introducing two more universes of agents: InitAgent (initializer agent) and OwnAgent (ownership agent).

**\(LC_0\): Processor Agents**

A processor agent is characterized by the attributes loc and proc: loc is the location on which the agent performs actions, and proc is the processor on behalf of which the agent acts. Both attributes have fixed values; thus they are modeled as static functions.

Associated with each ProcAgent are some attributes whose values may change during an execution, thus are modeled as dynamic functions. For instance, the attribute opType indicates the type of operation that the agent is to perform in the current step. Some operations may take multiple steps; for instance, a processor agent performing an acquire operation on a location may need to
Function | Profile
---|---
\( p\.loc \) | ProcAgent \( \rightarrow \) Location
\( p\.proc \) | ProcAgent \( \rightarrow \) Processor
\( p\.opType \) | ProcAgent \( \rightarrow \) OpType
\( p\.nextOpType \) | ProcAgent \( \rightarrow \) OpType
\( p\.waiting? \) | ProcAgent \( \rightarrow \) Boolean
\( p\.writeVal \) | ProcAgent \( \rightarrow \) Value

Table 1: Attributes of ProcAgents.

wait for another processor agent to release ownership of that location. When the current operation is completed, the processor agent updates its \( \text{opType} \) attribute.

The type of the next operation that the agent is to perform is given by the attribute \( \text{nextOpType} \), an \textit{external} (a.k.a. \textit{monitored}) function, whose interpretation is determined by the environment. In contrast, the attribute \( \text{opType} \) is updated by agents (and never by the environment), so it is called a \textit{dynamic internal} function (or \textit{controlled}) function.

The attribute \( \text{waiting?} \) (a controlled function) determines whether a processor agent is waiting for ownership of its location (as the result of an acquire operation). If a processor agent is unable to gain ownership immediately, it sets its \( \text{waiting?} \) attribute to \text{true}. Finally, the monitored function \( \text{writeVal} \), provides the value written by a write operation. This function is not used in \( LC_0 \) but is used in both \( LC_{mm} \) and \( LC_{cp} \).

In Table 1, we present the attributes for the processor agents with their profile (their signature).

\( LC_0 \): Initializer Agents

A question arises regarding the initial status of each location: If a processor agent reads from a location that has never been written to, it is not clear what the result should be. We avoid this complication by ensuring that each location is properly initialized before the processor agents start to perform operations on it. For each specific location, this task is handled by an \textit{InitAgent} ("initializer agent"). Since the details (attributes and transition rules) of these agents are straightforward, we omit them. Note that we assume that each location has an \textit{initialized?} attribute that is set to \text{true} once the appropriate \textit{InitAgent} has completed.

\( LC_0 \): Ownership Agents

A processor agent can gain ownership only if there are no other processor agent that currently owns the location. If another agent does own the location, the agent wishing to acquire must wait. There may be multiple processor agents waiting for ownership of the same location. The decision as to which agent is granted ownership is beyond the control of any processor agent and the arbitration policy is not of interest to our specification. Therefore we define the \textit{OwnAgent} universe whose members have the responsibility of arbitrating ownership of locations and we associate a unique \textit{OwnAgent} with each memory location.
Since each ownership agent deals with a single location, each OwnAgent has a (static) loc attribute. Each location also has an owner attribute, indicating which processor agent (if any) currently owns the location. When a processor agent releases a location, there may be other processor agents waiting to gain ownership. The monitored (oracle) function nextOwner indicates the next processor agent selected to receive ownership of the location. This monitored function is consulted by the OwnAgent. The profile for these attributes are presented in [Table 2].

**Terminology**

We introduce the following terminology for agents and actions in runs of any of the model.

**Definition** If a ProcAgent p makes a move Rd at which p.opType = read, we say that p performs a read (or simply reads) at Rd. (Similarly for write.)

**Definition** If a ProcAgent p makes a move A at which p.opType = acquire and p.loc.owner = p, we say that p performs an acquire (or simply acquires) at A. (Similarly for release.)

**LC0: Conditions on Runs**

Some aspects of our models LC0, LCmm, and LCcp are outside the control of the ASM transition rules. First, our static functions must have certain properties. We restrict attention to runs in which the following conditions are true of the static functions loc and proc:

**Static condition 1** For every Processor P and for every Location ℓ, there is a unique ProcAgent p for which p.proc = P and p.loc = ℓ.

**Static condition 2** For every Location ℓ, there is a unique InitAgent i for which i.loc = ℓ.

**Static condition 3** For every Location ℓ, there is a unique OwnAgent o for which o.loc = ℓ.

Second, certain conditions must be true in the initial state of any run. We restrict our attention to runs in which the following conditions are true initially:

<table>
<thead>
<tr>
<th>Function</th>
<th>Profile</th>
</tr>
</thead>
<tbody>
<tr>
<td>o.loc</td>
<td>OwnAgent → Location</td>
</tr>
<tr>
<td>ℓ.owner</td>
<td>Location → ProcAgent</td>
</tr>
<tr>
<td>ℓ.nextOwner</td>
<td>Location → ProcAgent</td>
</tr>
</tbody>
</table>

**Table 2:** Attributes of OwnAgents and Locations.
Init condition 1 For every Location $\ell$, $\ell$.owner.undef? and not $\ell$.initialized?.

Init condition 2 For every ProcAgent $p$, not $p$.waiting?.

Also, the monitored function nextOwner must produce “reasonable” values at every move of any run: Only a processor agent currently waiting to obtain ownership on the location should be granted ownership. Thus we restrict attention to runs in which the following condition is met at every move:

Run condition 1 For every Location $\ell$, if $\ell$.nextOwner.def?, then $\ell$.nextOwner.loc = $\ell$ and $\ell$.nextOwner.waiting?.

Finally, in order to remain faithful to Gao and Sarkar’s description, we restrict our attention to runs in which acquires and releases come in matching pairs.

Run condition 2 If a ProcAgent $p$ acquires at a move $A_p$ and releases after $A_p$, then there is a move $R_p$ after $A_p$ at which $p$ releases such that $p$ does not acquire in $(A_p, R_p)$.

Run condition 3 If a ProcAgent $p$ releases at a move $R_p$, then there is a move $A_p$ before $R_p$ at which $p$ acquires such that $p$ does not release in $(A_p, R_p)$.

$LC_0$: Processor Agent Module

The behavior of a processor agent is presented as an ASM module in [Figure 1], where the general behavior is as follows: based on the current opType, the actions specified by an appropriate abstract rule (Read, Write, Acquire, or Release) are performed. These rules are redefined in [Section 3], giving us a complete ASM model $LC_m$. A different set of definitions for these same rules then appears in [Section 4], resulting in a distinct ASM model $LC_p$.

A ProcAgent may begin performing operations only when its associated location has been initialized. While an operation is executed, the operation to be performed in the next step is obtained through the rule Get next operation, which simply consults the environment to determine what should be done next. Note that a processor agent may update its opType attribute to undef. In this case, it temporarily stops performing memory operations but continues to execute its program, executing Get next operation until the resulting opType is “well-defined” (non-undef).

The acquire case is slightly different because a processor agent must first wait for ownership of the location. Thus, execution of the ProcAgent module with opType = acquire does not change opType until the location has been acquired (i.e., Self.loc.owner = Self). As for the release case, it is Run Condition 3 that ensures that the releasing agent indeed has ownership of the location, so it is correct to release ownership (i.e., update Self.loc.owner to undef).
module ProcAgent:
if Self.loc.initialized? then
  case Self.opType of
    read:  Read
    write: Write
    acquire: Acquire
    release: Release
    undef:  Get next operation
  end
rule Read:
Get next operation

rule Write:
Get next operation

rule Acquire:
if Self.loc.owner ≠ Self then Self.waiting? := true
else Get next operation

rule Release:
Self.loc.owner := undef
Get next operation

rule Get next operation:
Self.opType := Self.nextOpType

Figure 1: Module for processor agents (ProcAgent).

module OwnAgent:
if Self.loc.owner.undef? and Self.loc.nextOwner.def? then
  Self.loc.nextOwner.waiting? := false
  Self.loc.owner := Self.loc.nextOwner

Figure 2: Module for ownership agents (OwnAgent).

LC0: Ownership Agent Module

The module for ownership agents is given in [Section 2]. If the location associated with the agent currently has no owner and nextOwner is defined, then according to Run Condition 1, the processor agent indicated by nextOwner is currently waiting to gain ownership of the location. Therefore the ownership agent grants ownership to the processor agent, updating its waiting? status to false and making it the owner. Note that the waiting? attribute's only role is to allow this interaction with the OwnAgent: once a ProcAgent updates its waiting? attribute to true, only the appropriate OwnAgent can update it to false. The same is true of the owner attribute: once an OwnAgent updates it to a particular ProcAgent p, only that ProcAgent can change it (when releasing the location).
3 The LC Memory Model

In the previous section, we described a generic framework — in terms of abstract Read, Write, Acquire and Release rules — that provides the top-level description of both the memory consistency model (LC\textsubscript{mm}) and the cache protocol model (LC\textsubscript{cp}). In this section we present a complete specification for the LC model, LC\textsubscript{mm}, by defining the transition rules according to the memory model specifications.

The state of a memory system is determined entirely by the operations that have been performed upon the system. Following [Gao and Sarkar 00], we view the state of a memory system as a history of events (i.e., instances of operations) that modify the memory system state. These events are organized according to a partial order relation. The following information is recorded for each event: its type (read, write, acquire, release), the agent that generated it (its issuer), and the location on which it was performed.

Events are temporally ordered by a relation ≺. Each processor must act as if it observed the events in an order compatible with ≺. When a processor performs an operation, an event is added to the history, and ≺ is updated accordingly. In practice, the memory system does not maintain such a history, but this view is useful for thinking of consistency models in an implementation-independent way. How ≺ is updated depends on the consistency model adopted. For instance, SC would require a total order of events, common to all processors. On the other hand, in a relaxed model like LC, a partial order is sufficient.

For any memory model, a key question is: what value should be returned when a processor performs a read? For a strong memory model, there is a unique value to be returned, the value written by the most recent write operation. However, when a weaker memory model is used, there may be more than one value associated with a single location at a given time. In such models, a read operation is thus associated with a set of values.

A specification of a memory consistency model can thus be characterized by two main features:

- What is the precedence relation between a new event and other events already recorded in the history?

  In the case of LC, this question is answered as follows. A new write, acquire, or release event $e$ by a processor agent $p$ on a location $\ell$ is ordered so that it succeeds any event already issued by $p$ on $\ell$. In other words, the set of events by $p$ on $\ell$ is linearly ordered. Furthermore, since the history is a partial order and ≺ is transitive, the new event also succeeds any event $e' ≺ e$, including events issued by other processor agents.

  In the case of a new acquire event $a$, the partial order is updated further. The latest release event issued on $\ell$ (by any processor agent)\textsuperscript{2} precedes $a$, along with any events that precede that release. This release could have been issued by any processor agent, not necessarily the issuer of the new acquire. Hence, it is through acquires that events issued by different processor agents are ordered.

\textsuperscript{2} Note that there is at most one latest release for $\ell$ at any given time, since (as pointed out in [Section 2] and formalized by Run Condition 3) a processor agent only releases $\ell$ if it has (exclusive) ownership of $\ell$. Moreover, for each location the initializer agent also issues a release.
Which values are associated with a new read event?
In LC, when a processor agent $p$ issues a read on a location $\ell$, any write event on $\ell$ that has not been “overwritten” by another write event has its value associated with the new read event. We formalize this notion as follows. Let $e$ be the last event issued by $p$; then according to the LC model, write event $w$ is readable by $p$ if and only if there is no write event $w'$ such that $w \prec w' \preceq e$. This can be true of a write event $w$ in either of the following ways:

- If $w$ precedes $e$ and $w$ is “recent” in the sense that there is no intervening write event between $w$ and $e$, $w$’s value is readable.
- Alternatively, if $w$ is simply unordered with respect to $e$, $w$’s value is also readable.\(^3\)

Our specification differs from Gao and Sarkar’s description in a few respects. First, we model a read as a single-step operation and we do not place read events in the history. Second, our rules ensure that $\prec$ remains a transitive relation throughout the course of the system’s execution.

\(\text{LC}_{mm}:\) Universes, Attributes, and Relations

<table>
<thead>
<tr>
<th>Function</th>
<th>Profile</th>
</tr>
</thead>
<tbody>
<tr>
<td>$e$.issuer</td>
<td>Event $\rightarrow$ ProcAgent $\cup$ InitAgent</td>
</tr>
<tr>
<td>$w$.val</td>
<td>WriteEvent $\rightarrow$ Value</td>
</tr>
<tr>
<td>$p$.latestEvent</td>
<td>ProcAgent $\rightarrow$ Event</td>
</tr>
<tr>
<td>$\ell$.latestRelease</td>
<td>Location $\rightarrow$ ReleaseEvent</td>
</tr>
<tr>
<td>$i$.initWrite</td>
<td>InitAgent $\rightarrow$ WriteEvent</td>
</tr>
<tr>
<td>reads?($rd, v$)</td>
<td>ReadEvent $\times$ Value $\rightarrow$ Boolean</td>
</tr>
<tr>
<td>$e \prec e'$</td>
<td>Event $\times$ Event $\rightarrow$ Boolean</td>
</tr>
</tbody>
</table>

Table 3: Additional attributes and relations for \(\text{LC}_{mm}\).

We define universes \texttt{ReadEvent}, \texttt{WriteEvent}, \texttt{AcquireEvent} and \texttt{ReleaseEvent} to represent the sets of events of various types, and the universe \texttt{Event} to refer to the union of these various sets. Each \texttt{Event} has an \texttt{issuer} attribute (the agent that issued the event). A \texttt{WriteEvent} also has a \texttt{val} attribute indicating the value written.

We introduce attributes to maintain the most recent events issued. Each processor agent has a \texttt{latestEvent} attribute (the most recent event issued by the agent) and each location has a \texttt{latestRelease} attribute (the most recent release issued on the location).

Finally, we define two key relations, which are both empty initially:

\(^3\) Note that if $w$ is unordered with respect to $e$, then the associated write has been performed by another processor agent $q$, and $p$ and $q$ have not synchronized with proper acquire/release operations. Thus the value of $w$ could have been written to memory at an arbitrary moment, which is why it must be considered readable by $p$.\(^\text{1097}\)
reads?(rd, v) indicates whether value v can be read at ReadEvent rd. The set of values that can be read by rd is thus \{v|reads?(rd, v)\}.

- \( e \prec e' \) represents the partial order among memory events.

Attributes and relations associated with events and with locations are presented in [Table 3].

**Terminology**

The following terms refer to the issuing of events in a run of \( LC_{mm} \).

**Definition** An event \( e \) with \( e.issuer = p \) (for some \( \text{ProcAgent} \ p \)) is a \( p \)-event.

**Definition** If a \( \text{ProcAgent} \ p \) makes a move \( Rd \) that creates a ReadEvent \( rd \), we say that \( p \) issues a read event \( rd \) at \( Rd \). (Similarly for write, acquire, and release.)

**Definition** If a \( \text{ProcAgent} \ p \) reads at a move \( Rd \) and readOK?(w, p) for a WriteEvent \( w \), we say that \( p \) reads \( w \) at \( Rd \). We also say that \( p \) reads value \( w.val \) at \( Rd \).

**\( LC_{mm} \): Conditions on Runs**

We restrict attention to runs in which the following conditions are true in the initial state of \( LC_{mm} \):

**Init condition 3** For every Location \( \ell \), \( \ell .latestRelease.undef? \).

**Init condition 4** For every \( \text{ProcAgent} \ p \), \( p .latestEvent.undef? \).

**\( LC_{mm} \): Terms and Transition Rules**

The rules for the non-read operations by processor agents in \( LC_{mm} \) are given in [Figure 3], where in each case a new event of the appropriate type is created whose issuer is Self, i.e., the agent that executes the rule and generates the event.

The rule for read operations is given in [Figure 4]. The term readOK?(w, p), also defined in [Figure 4], determines whether the write value of WriteEvent w is readable for \( \text{ProcAgent} \ p \). For the value of WriteEvent w to be readable by processor agent \( p \) at a move \( Rd \), w must be a write to the appropriate location, and as noted earlier, there must be no WriteEvent \( w' \) that “overwrites” \( w \).

The set of values that can be read by a new ReadEvent is specified by updating the reads? relation. Any write event whose value is considered readable (according to readOK?) is in the set. For all non-read events, \( \prec \) is updated to account for the newly created event:

- The new event succeeds its issuer’s latest event (as well as all predecessors of that event).
rule Write:
extend WriteEvent with w
    w.issuer := Self
    w.val := Self.writeVal
    Order w after Self.latestEvent and its predecessors
    Self.latestEvent := w
Get next operation.

rule Acquire:
if Self.loc.owner ≠ Self then Self.waiting? := true
else
    extend AcquireEvent with a
        a.issuer := Self
        Order a after Self.latestEvent and its predecessors
        Order a after Self.loc.latestRelease and its predecessors
        Self.latestEvent := a
    Get next operation

rule Release:
extend ReleaseEvent with r
    r.issuer := Self
    Order r after Self.latestEvent and its predecessors
    Self.latestEvent := r
    Self.loc.latestRelease := r
    Self.loc.owner := undef
Get next operation

rule Order e after d and its predecessors:
if d.def? then
    d ≺ e := true
    do-forall c: Event: c ≺ d
        c ≺ e := true

Figure 3: $LC_{mm}$ rules for write, acquire and release operations.

Synchronization between processors imposes additional ordering constraints. In LC, these synchronizations occur exclusively through acquire and release operations. Thus a new AcquireEvent succeeds the latest release event on the location being acquired — which, by Run Conditions 2 and 3, is sure to exist and is sure to have been performed by the appropriate ProcAgent — as well as all predecessors of the latest release.

The rules presented in Figs. 3–4 refine the processor agent modules of $LC_0$. Along with the ownership agent module in [Figure 2], they complete $LC_{mm}$, our ASM representation of the LC model.

4 The LC Cache Protocol

We now present $LC_{cp}$, a formal model of the LC cache protocol, in which we make various assumptions about how values can be stored. In particular, we
**rule** Read:
**extend** ReadEvent with rd
    rd.issuer := Self
    do-forall w: WriteEvent: readOK?(w, Self)
        reads?(rd, w.val) := true
**Get next operation**

**term** readOK?(w, p):
    w.issuer.loc = p.loc and not (∃w': WriteEvent) w ≺ w' ⪯ p.latestEvent

*Figure 4: Rule and auxiliary term for read operation in LCm*.

Assume that each processor is equipped with its own cache and that there is a set of memory cells collectively called main memory, distinct from any processor’s cache. Each location has a value stored in main memory. Processors store temporary copies of values from main memory in their caches. The processors can modify these copies without necessarily updating main memory. When a processor writes to a location, the new value is written to the processor’s cache. Eventually this value is also written back to the main memory. Thus, in this model, agents update cache entries and main memory locations instead of a history of events.

At any time, each cache entry is either valid or invalid, and a valid entry is either clean or dirty. A valid entry has a readable value, while an invalid one does not. A clean entry has a value from main memory that has not been overwritten; a dirty entry has a value written by the local processor that has not been written back to the main memory. When all the cache entries are occupied, a write or read of a location with no entry in the cache requires the removal (or ejection) of an existing location from the cache. A cache replacement policy is used to select which location should be removed from the cache.

A writeback to main memory is not a single-step action. There is some delay between the initiation of a writeback (when the value stored in the cache is sent to the memory) and the completion of the writeback (when the value is finally recorded in memory). Writebacks may be completed concurrently with actions by processor agents. To represent the process of writing back values to main memory, we introduce a universe of writeback agents. Any writeback is initiated by generating a writeback agent and by copying the dirty cached value to the writeback agent. The writeback is completed when the writeback agent copies this value to main memory.

Our view of writebacks as multi-step actions requires us to clarify the meaning of a release operation. One effect of a release is to make the last write by the releaser available to other processor agents. This is why a release initiates a writeback in the case of a dirty cache entry. But since a writeback cannot be performed in a single step, the following question arises: is it sufficient to initiate the writeback before completion of the release (i.e., give up ownership and proceed to the next operation), or must the writeback also be completed? Gao and Sarkar indicate the latter. This assumption implies that a releasing processor agent has to wait for a writeback to complete before proceeding to the
next operation [Gao and Sarkar 00].

The actions for each operation are as follows. When a processor agent issues a read to a location that has no entry in the cache, the read will add a value to the cache. If the location’s most recent writeback agent has a value whose write back operation has not yet completed, the value of that writeback is added to the cache. Otherwise the value stored in main memory is added to the cache. A write generates a value, caches it, and updates the status of the cache entry to dirty. An acquire of a location invalidates the cache entry for the location, unless the entry is dirty (in which case the last value written by the processor remains in the cache because it is a legal value for subsequent read operations). A release of a dirty location initiates a writeback of the value stored in the cache, then waits until the value is transferred to main memory. Only when that writeback and all previous ones to the same location are completed does the release terminate.

Note that the LC cache protocol only requires two inexpensive operations to enable synchronization between multiple processors: the self-invalidation of cache entries that are not dirty for the acquire rule, and the writeback of a dirty cache entry for the release rule. Therefore no expensive invalidation or update requests need to be sent across the network under the LC cache protocol.

<table>
<thead>
<tr>
<th>Function</th>
<th>Profile</th>
</tr>
</thead>
<tbody>
<tr>
<td>p.cacheVal</td>
<td>ProcAgent → Value</td>
</tr>
<tr>
<td>p.cacheValid?</td>
<td>ProcAgent → Boolean</td>
</tr>
<tr>
<td>p.cacheDirty?</td>
<td>ProcAgent → Boolean</td>
</tr>
<tr>
<td>p.ejectee</td>
<td>ProcAgent → ProcAgent</td>
</tr>
<tr>
<td>p.latestWB</td>
<td>ProcAgent → WritebackAgent</td>
</tr>
<tr>
<td>ℓ.MMVal</td>
<td>Location → Value</td>
</tr>
<tr>
<td>wb.issuer</td>
<td>WritebackAgent → ProcAgent</td>
</tr>
<tr>
<td>wb.val</td>
<td>WritebackAgent → Value</td>
</tr>
<tr>
<td>wb.active?</td>
<td>WritebackAgent → Boolean</td>
</tr>
</tbody>
</table>

Table 4: Additional attributes for LCcp.

**LCcp:** Attributes

In LCcp, a processor agent p is also associated with a processor P and a location ℓ. For each processor agent p, the attribute cacheVal gives the value in p’s cache for location ℓ (if any such value exists), and cacheValid? and cacheDirty? give the valid/invalid status and dirty/non-dirty status of the cache entry.

In order not to tie our model to any specific cache replacement policy, the cache entry to be ejected (if any) is determined by a monitored function, the attribute ejectee. For each processor agent p, ejectee selects another processor agent for the same processor which also has a cache entry; the cache entry of p.ejectee is then to be ejected in order to make room for p’s entry.

The attribute MMVal is associated with each location ℓ and represents the value currently stored in the main memory for ℓ. The new universe WritebackAgent represents the agents charged with writing values to main memory. The attribute
latestWB associated with each ProcAgent \( p \) gives the writeback agent most recently generated by \( p \). We associate three attributes with the WritebackAgent universe: issuer, which gives the processor agent that generated the writeback agent; val, which gives the value to write to main memory; and active?, which determines whether a given writeback agent has yet to write its value to main memory. [Table 4] summarizes the attributes used to model caches, writeback agents, and the main memory.

**Terminology**

In \( LC_{cp} \), releases are multi-step actions. Therefore, we must reformulate what it means for a processor agent to perform a release. In our terms, a processor agent first prepares to perform a release by initiating a writeback of its dirty cache entry and waiting for the writeback to complete. It only performs the release (relinquishing ownership) after these actions have completed. We formalize this as follows.

**Definition** If a ProcAgent \( p \) makes a move \( R_p \) at which \( p.\text{opType} = \text{release} \), then

- If \( p.\text{cacheDirty}? \) or not \( p.\text{allWritebacksCompleted}? \), we say that \( p \) prepares to release at \( R_p \);
- Otherwise, we say that \( p \) releases at \( R_p \).

We use the following terms to characterize read actions and cache maintenance actions in \( \rho_{\text{cp}} \).

**Definition** If a ProcAgent \( p \) reads at a move \( Rd_p \), we say that \( p \) reads value \( v \) at \( Rd_p \), where \( v = p.\text{cacheVal} \) if \( p.\text{cacheValid}? \) and \( v = p.\text{loc.MMVal} \) otherwise.

**Definition** Let \( p \) be a ProcAgent that reads at a move \( Rd_p \).

- If not \( p.\text{cacheValid}? \) and \( p.\text{allWritebacksCompleted}? \), we say that \( p \) performs a miss read at \( Rd_p \);
- Otherwise, if \( p.\text{cacheDirty}? \) or not \( p.\text{allWritebacksCompleted}? \), we say that \( p \) performs a dirty read at \( Rd_p \);
- Otherwise, we say that \( p \) performs a clean read at \( Rd_p \).

**Definition** Let \( p \) be a ProcAgent, and let \( wb_p \) be a WritebackAgent for which \( wb_p.\text{issuer} = p \).

- If at a move \( I_p \), \( p.\text{cacheDirty}? \) is updated from true to false, we say that a writeback of \( p \)’s cache entry is initiated at \( I_p \).
- If at a move \( C_p \), \( wb_p.\text{active}? \) is updated from true to false, we say that a writeback of \( p \)’s cache entry is completed at \( C_p \).
- Let \( I_p \) be a move at which a writeback of \( p \)’s cache entry is initiated and \( wb_p \) is generated. Let \( C_p \) be a move of \( wb_p \) at which a writeback of \( p \)’s cache is completed. Then we say that the writeback initiated at \( I_p \) is completed at \( C_p \).

\[4\] Note that a writeback may be initiated by \( p \) itself (through a release) or by another ProcAgent (through a read or write that triggers an ejection of \( p \)’s cache entry).
**LC<sub>cp</sub>: Conditions on Runs**

We put the following restrictions on initial states of $LC_{cp}$.

**Init condition 5** For every ProcAgent $p$, not ($p$.cacheValid? or $p$.cacheDirty?).

**Init condition 6** The WritebackAgent universe is empty.

The attribute ejectee must take on reasonable values during a run. We restrict attention to runs that obey the following conditions:

**Run condition 4** For every ProcAgent $p$, if $p$.ejectee.def?, then $p$.ejectee.proc = $p$.proc and $p$.ejectee.cacheValid?.

**Run condition 5** For every ProcAgent $p$, if $p$.ejectee.def?, then $p$.ejectee.opType ≠ read and $p$.ejectee.opType ≠ write.

**LC<sub>cp</sub>: Transition Rules**

---

**rule** Eject cache entry of $p$:

$p$.cacheValid? := false

if $p$.cacheDirty? then

  Initiate writeback on cache entry of $p$

**rule** Initiate writeback on cache entry of $p$:

$p$.cacheDirty? := false

extend WritebackAgent with $wb_p$

$wb_p$.issuer := $p$

$wb_p$.val := $p$.cacheVal

$wb_p$.active? := true

$p$.latestWB := $wb_p$

---

**module** WritebackAgent:

if Self.active? then

  Self.loc.MMVal := Self.val
  Self.active? := false

---

**term** $p$.allWritebacksCompleted?:

($\forall wb_p$: WritebackAgent: $wb_p$.issuer = $p$) not $wb_p$.active?

---

**Figure 5** $LC_{cp}$ rules for cache maintenance.

The rules and terms associated with cache ejection and writeback are presented in [Figure 5]. The ejection of a cache entry requires an invalidation of
the cache entry, and a writeback if the entry is dirty. The writeback initiation
updates the cache entry’s status to non-dirty, generates a writeback agent, and
passes the cached value to the writeback agent. A writeback agent makes a single
move in which it copies its value to main memory.

\[
\text{rule Read:}
\]
\[
\text{if not Self.cacheValid? then}
\]
\[
\text{if Self.allWritebacksCompleted? then Self.cacheVal := Self.loc.MMVal}
\]
\[
\text{else Self.cacheVal := Self.latestWB.val}
\]
\[
\text{Self.cacheValid? := true}
\]
\[
\text{if Self.ejectee.def? then Eject cache entry of Self.ejectee}
\]

\[\text{Get next operation}\]

\[
\text{rule Write:}
\]
\[
\text{Self.cacheVal := Self.writeVal}
\]
\[
\text{Self.cacheValid? := true}
\]
\[
\text{Self.cacheDirty? := true}
\]
\[
\text{if Self.ejectee.def? then Eject cache entry of Self.ejectee}
\]

\[\text{Get next operation}\]

\[
\text{rule Acquire:}
\]
\[
\text{if Self.loc.owner \neq Self then Self.waiting? := true}
\]
\[
\text{else}
\]
\[
\text{if Self.cacheValid? and not Self.cacheDirty? then Self.cacheValid? := false}
\]

\[\text{Get next operation}\]

\[
\text{rule Release:}
\]
\[
\text{if Self.cacheDirty? then Initiate writeback on cache entry of Self}
\]
\[
\text{elseif Self.allWritebacksCompleted? then}
\]
\[
\text{Self.loc.owner := undef}
\]

\[\text{Get next operation}\]

Figure 6: \(LC_{cp}\) rules for read, write, acquire and release operations by processor agents.

The rules for read, write, acquire, and release operations by processor agents
are presented in [Figure 6]. If there is no valid cache entry, reading involves
fetching a value from the last writeback agent or from main memory. Writing
involves storing a new value in the cache.

In the case of a read or write, a new cache entry may be needed; therefore the
attribute ejectee is checked to determine whether a cache entry is to be ejected
to make room for the new one. The rules for acquire and release operations are
simple. An acquire invalidates a clean cache entry. A release initiates a writeback
of the cache entry, if it is dirty. Only when all writebacks on the cache entry are
completed does the release terminate.
In this section, we outline the proof that shows that the cache protocol described by LC\textsubscript{cp} implements the abstract model described by LC\textsubscript{mm}. For a detailed proof, the reader should consult our technical report [Wallace \textit{et al.} 01]. More precisely, our goal is to show that any value read in an execution of LC\textsubscript{cp} is also a legal value in an equivalent execution of LC\textsubscript{mm}. In a run of LC\textsubscript{mm}, for each read operation a set of legal readable values is computed, while in the run of LC\textsubscript{cp} a single value is read at each read operation. We consider runs of LC\textsubscript{mm} and LC\textsubscript{cp} in which the memory operations that are performed and the order in which they are performed are identical. We then show that for each read operation of LC\textsubscript{cp}, the single value read is in the set of readable values computed at the corresponding move of LC\textsubscript{mm}'s run.

**Equivalent Runs of LC\textsubscript{mm} and LC\textsubscript{cp}**

We must first start by considering what it means for runs of LC\textsubscript{mm} and LC\textsubscript{cp} to be equivalent. An ASM run consists of a partial order of moves performed by agents, with some agent executing its associated module at each move. Informally, for runs of the two models to be equivalent, the system components (locations and processors) must be the same, and the same agents must make the same moves in the same order. More precisely, the following conditions must be met:

- The static information (e.g., number of processors, locations, and agents) must be the same in the two runs.
- The runs must have the same partial order of moves.
- For each move, the environment in the two runs must produce the same results for the monitored functions \texttt{nextOpType}, \texttt{writeVal}, and \texttt{nextOwner}.

We formalize the above as follows:

- Let \( \sigma \) be a state of LC\textsubscript{mm} or LC\textsubscript{cp}. Then \( \sigma^- \) is the reduct of \( \sigma \) to the static and monitored functions common to LC\textsubscript{mm} and LC\textsubscript{cp} (i.e., the static and monitored functions of LC\textsubscript{0}, introduced in [Section 2]).
- A state \( \sigma_{\text{mm}} \) of LC\textsubscript{mm} is equivalent to a state \( \sigma_{\text{cp}} \) of LC\textsubscript{cp} if \( \sigma_{\text{mm}}^- \) and \( \sigma_{\text{cp}}^- \) are isomorphic.

Let \( \rho_{\text{cp}}^* = (\mu_{\text{cp}}^*, \alpha_{\text{cp}}, \sigma_{\text{cp}}^*) \) be a run of LC\textsubscript{cp}. \( \mu_{\text{cp}}^* \) is a partially ordered set of moves, \( \alpha_{\text{cp}} \) is a function mapping moves to agents (i.e., gives the agent performing a move), and \( \sigma_{\text{cp}}^* \) is a function mapping finite initial segments of moves to states of LC\textsubscript{cp} (the state resulting from each finite initial segment of moves).

In the proofs, we need to consider a sequential “equivalent” of a distributed run rather than the run itself. According to the ASM Lipari guide [Gurevich 95], we lose no generality by proving correctness of an arbitrary linearization of a run. Hence, we can consider a linearization \( \rho_{\text{cp}} = (\mu_{\text{cp}}, \alpha_{\text{cp}}, \sigma_{\text{cp}}) \) of \( \rho_{\text{cp}}^* \). \( \mu_{\text{cp}} \) is a topological sort of \( \mu_{\text{cp}}^* \); a linearly ordered set of moves (an arbitrary interleaving) that has the same moves as \( \mu_{\text{cp}}^* \) and preserves all the ordering of \( \mu_{\text{cp}}^* \).

Let \( \rho_{\text{mm}} = (\mu_{\text{mm}}, \alpha_{\text{mm}}, \Sigma_{\text{mm}}) \) be a run of LC\textsubscript{mm} that is equivalent to a run of LC\textsubscript{cp}, as defined below. Informally, in the runs \( \rho_{\text{mm}} \) and \( \rho_{\text{cp}} \), the same agents
perform the same operations in the same order; only the implementation details
differ: in $LC_{mm}$ the partial order $\prec$ is updated, while in $LC_{cp}$ it is the cache
entries and main memory locations that are updated.

More formally, a run $\rho_{mm}$ can be considered equivalent to a run $\rho_{cp}$ as follows.
First of all, it should be noted that fewer moves are made in $\rho_{mm}$ than in $\rho_{cp}$.
WritebackAgents do not exist in $\rho_{mm}$ and so do not make moves; a release in
$\rho_{mm}$ is always a single-move action (there is no need to prepare for a release).
We thus restrict $\mu_{mm}$ to the moves of $\mu_{cp}$ that are neither writeback-agent moves
nor release preparation moves. More formally, $\mu_{mm} = \mu_{cp} \setminus (WB \cup PR)$, where
$WB = \{ M \in \mu_{cp}.WritebackAgent(\alpha_{cp}(M)) \}$ and
$PR = \{ M \in \mu_{cp}.\alpha_{cp}(M) \text{ prepares to release at } M \}$. Likewise, we define $\alpha_{mm}$ as
the restriction of $\alpha_{cp}$ to moves of $\mu_{mm}$. Finally, for each prefix $X$ of $\mu_{mm}$,
$\sigma_{mm}(X)$ is equivalent to $\sigma_{cp}(X)$. Since the only sources of nondeterminism in
$LC_{mm}$ are the monitored functions $\text{nextOpType}$, $\text{writeVal}$ and $\text{nextOwner}$, and
these are identical in $\rho_{cp}$ and $\rho_{mm}$, $\rho_{mm}$ is unique up to isomorphism.

**Lemmata: Ordering of Events in $\rho_{mm}$**

The proof that $LC_{cp}$ implements $LC_{mm}$ hinges on two important properties of
the ordering of events in $\rho_{mm}$, stated in the following lemmata. Lemma 1 states
that the events issued by a ProcAgent are linearly ordered by $\prec$; whenever a processor agent issues two events $d$ and $e$ in sequence, $d$ becomes a predecessor
of $e$. This can be proved using a straightforward induction on the number of
$p$-events issued between moves $D_p$ and $E_p$.

**Lemma 1** In $\rho_{mm}$, let $p$ be a ProcAgent, let $d_p$ be a $p$-event issued at a move
$D_p$, and let $e_p$ be a $p$-event issued at a move $E_p$ after $D_p$. Then $d_p \prec e_p$.

The next property concerns how events issued by different agents in $\rho_{mm}$ can
become ordered with respect to each other. In determining whether a write event
$w_p$ by one agent $p$ is readable by another agent $q$ (assuming that $p$ and $q$ operate
on a common location), it is necessary to determine whether $w_p$ precedes $q$’s
latest event (according to $\prec$). If not, $w_p$ is readable; if so, $w_p$ is only readable
if there is no write event intervening between $w_p$ and $q$’s latest event. Lemma 2
asserts that
- a $p$-write becomes a predecessor of a $q$-event if $p$ releases after the write and
$q$ then acquires;
- this is the only way that a $p$-write can come to be ordered with respect to a
$q$-event.

**Lemma 2** In $\rho_{mm}$, let $p$ and $q$ be distinct ProcAgents for which $p.\text{loc} = q.\text{loc}$.
Let $W_p$ be a move at which $p$ issues a WriteEvent $w_p$, and let $R_d_q$ be a move
after $W_p$ at which $q$ issues a ReadEvent. Then $w_p \prec q.\text{latestEvent}$ at $R_d_q$ if and
only if
- $p$ issues a ReleaseEvent $r_p$ at a move $R_p$ in the interval $(W_p, R_d_q)$ and
- $q$ issues an AcquireEvent $a_q$ at a move $A_q$ in the interval $(R_p, R_d_q)$. 
Lemmata: Properties of Read Operations

Lemmata 3–5 concern the three types of read operation in $\rho_{cp}$: dirty, miss, and clean. For a read operation of any type in $\rho_{cp}$, we establish that the value read is one of the (possibly many) values read at the corresponding move of $\rho_{mm}$.

**Lemma 3** In $\rho_{cp}$, let $Rd^D_p$ be a move at which a ProcAgent $p$ performs a dirty read of a Value $v$. Then in $\rho_{mm}$, $p$ also reads $v$ at $Rd^D_p$.

**Proof outline.** Let $\ell$ be the location associated with $p$. A dirty read at $Rd^D_p$ means that, in $\rho_{cp}$, $p.cacheDirty?$ or $\neg p.allWritebacksCompleted$ at the time the read is performed. By Lemma 2, this last write is unreadable by $p$ in $\rho_{mm}$ if and only if, in the interval $I$ between the write and the read, (1) $p$ releases its location $\ell$; and then (2) some other processor agent acquires $\ell$, writes, and then releases; and then (3) $p$ acquires. However, a release by $p$ cannot complete in the interval $I$ since preparing for a release always sets $p.cacheDirty?$ to false and completing the release means $p.allWritebacksCompleted$ is true, thus contradicting the fact that the read at $Rd^D_p$ was dirty. Therefore, $v$ is a readable value in $\rho_{mm}$. $\blacksquare$

**Lemma 4** In $\rho_{cp}$, let $Rd^M_p$ be a move at which a ProcAgent $p$ performs a miss read of a Value $v$. Then in $\rho_{mm}$, $p$ also reads $v$ at $Rd^M_p$.

**Proof outline.** Let $\ell$ be the location associated with $p$. A miss read at $Rd^M_p$ means that, in $\rho_{cp}$, $\neg p.cacheValid?$ and $p.allWritebacksCompleted$. In other words, $p$ reads the last value of $\ell$ written back to main memory at some move $W$ by a processor agent $q$.

If $p = q$, the write at $W$ is unreadable in $\rho_{mm}$ only if $p$ performs a subsequent write before the read. However, if such a write was performed, it would have made the cache dirty, thus contradicting the fact that $Rd^M_p$ is a miss read (and the fact that the write at step $W$ was the last such write). Thus, the write at $W$ is readable in $\rho_{mm}$.

If $p \neq q$, Lemma 2 implies that the write at $W$ is unreadable in $\rho_{mm}$ if and only if, between $W$ and the read at $Rd^M_p$, (1) $q$ releases and then (2) $p$ acquires, and one of the following happens between $q$’s release and $p$’s acquire: (3a) $q$ writes again before releasing; or (3b) $p$ writes after acquiring; or (3c) some other processor agent acquires $\ell$, writes, and then releases. Cases (3a) and (3c) contradict the fact that $W$ was the last write written back to main memory. Case (3b) would have made the cache dirty; thus the read at $Rd^M_p$ would not be a miss read.

Therefore, $v$ is a readable value in $\rho_{mm}$. $\blacksquare$

**Lemma 5** In $\rho_{cp}$, let $Rd^C_p$ be a move at which a ProcAgent $p$ performs a clean read of a Value $v$. Then in $\rho_{mm}$, $p$ also reads $v$ at $Rd^C_p$.

**Proof outline.** A clean read at $Rd^C_p$ means that, in $\rho_{cp}$, $p.cacheValid?$, not $p.cacheDirty?$ and $p.allWritebacksCompleted$. In turn, this implies that $p$
reads a value from the cache, and that this value was placed there as a result of either (A) a miss read of a value written (at \( W \)) by some \( q \) or (B) the last write by \( p \).

In case (A), Lemma 2 implies that the write is unreadable in \( \rho_{mm} \) if and only if, between the write and the read, (1) \( q \) releases the location \( \ell \) and then (2) \( p \) acquires, and one of the following happens: (3a) \( q \) writes again before releasing; or (3b) \( p \) writes after acquiring; or (3c) some other processor agent acquires \( \ell \), writes, and then releases. (2) is not possible since it would invalidate the cache entry, an impossibility since the cache entry is valid at \( R_{Cp} \).

In case (B), Lemma 2 implies that the write is unreadable in \( \rho_{mm} \) if and only if (1) \( p \) releases and then (2) \( p \) acquires, and, between \( p \)'s release and \( p \)'s acquire, (3) some other processor agent acquires \( \ell \), writes, and then releases. But \( p \)'s acquire would invalidate the cache entry, which is impossible since the cache entry is valid at \( R_{Cp} \).

In either case, \( v \) is a readable value in \( \rho_{mm} \). ✷

**Theorem:** \( LC_{cp} \) Obeys \( LC_{mm} \)

Finally, we can state our main theorem, which follows directly from the previous lemmata:

**Theorem 1** Let \( R_{dp} \) be a move of \( \rho_{cp} \) at which a ProcAgent \( p \) reads a Value \( v \). Then at \( R_{dp} \) in \( \rho_{mm} \), \( p \) also reads \( v \).

6 \( LC_{cp} \) is Strictly Stronger Than \( LC_{mm} \)

In this section, we want to show that \( LC_{cp} \) does not allow certain behavior allowed by \( LC_{mm} \). In particular, we give an execution of \( LC_{mm} \) in which a particular value is read, value which cannot be read in any equivalent run of \( LC_{cp} \).

Consider a run \( \rho_{mm} \) of \( LC_{mm} \) with the following properties. In \( \rho_{mm} \), two distinct ProcAgents \( p \) and \( q \) operate on a common Location \( \ell \), and no other ProcAgents operate on \( \ell \). (Other ProcAgents may perform operations on Locations other than \( \ell \).) The operations of \( p \) and \( q \) occur in the following sequence:

- \( A_p \): Acquire by \( p \).
- \( W_p \): Write by \( p \), that writes the value 1.
- \( R_p \): Release by \( p \).
- \( W_q \): Write by \( q \), that writes the value 2.
- \( A_q \): Acquire by \( q \).
- \( R_{d_q} \): Read by \( q \).

At \( R_{d_q} \), the value 1 is readable according to \( LC_{mm} \), that is, in \( \rho_{mm} \), \( q \) can read the value 1 at move \( R_{d_q} \). The value 1 is unreadable at \( R_{d_q} \) only if it is overwritten by a write operation that is a successor of \( W_p \). Such a write cannot exist because: (1) there are no other writes by \( p \); (2) the only acquire is at \( A_q \); and (3) there are no writes after \( A_q \).

Then, we show that in any equivalent run of \( LC_{cp} \), 2 will definitely be the (sole) value read at \( R_{d_q} \). There are two cases:
1. If q’s cache entry is written back after the write (due to an ejection), 2 is written to main memory. By this time, the value 1 written by p must have already been written back before q’s write, since p releases before W_q. So 1 is overwritten by 2 in main memory; since there are no other writes to ℓ, there can be no further writebacks. Therefore q reads 2 from main memory at Rd_q.

2. If q’s cache entry is not written back, the value 2 is still present in the cache at Rd_q, so q reads 2 from the cache at Rd_q.

From these two properties, we can state the following theorem.

**Theorem 2** There exists a run ρ_{mm} of LC_{mm} in which a read operation Rd returns a value v that cannot be returned by the same read operation in any equivalent run ρ_{cp} of LC_{cp}.

7 Related work

There has been a substantial amount of research on the use of formal methods in the area of shared memory models. Much of this work focuses either on high-level memory models or on low-level cache consistency protocols. On the high-level side, Adve and Hill [Adve and Hill 93] define, in a semi-formal manner, two shared-memory models (data-race-free-0 and data-race-free-1) that attempt to capture the common features of various other weak memory models (e.g., weak ordering, release consistency). Frigo and Luchangco [Frigo and Luchangco 98] formally define various memory models using what they call a “computation-centric” approach. In this approach, a computation is defined in a manner similar to the abstract interpreter of Gao and Sarkar, using a directed acyclic graph that relates program operations. The effect of a memory model is defined through the specification of a set of observer functions that determine, for any given computation, which values can be read. This combination of computation and observer functions makes it possible to relate different memory models in a formal and simple way. Shen et al. [Shen et al. 99] introduce the CRF model, which attempts “to decompose load and store instructions into finer-grain orthogonal operations”. Using these instructions, for which they provide a formal term rewriting system, they then specify various other memory models. However, no equivalence proofs are really provided, since the specifications are really definitions by translation into CRF. Finally, the memory model for the Java programming language has inspired several efforts in formalization [Gontmakher and Schuster 00, Gurevich et al. 00, Maessen et al. 00, Manson and Pugh 01]; all of these works focus exclusively on the abstract memory model rather than particular implementations of it.

On the low-level side, various cache coherence protocols have been formally verified, usually relative to SC [Clarke et al. 94, Pong and Dubois 97] [Henzinger et al. 99]. Some of these proofs can be done automatically using (finite-state) model checking. How to formalize the LC cache coherence protocol in the context of model checking is not clear, since the partial order relation introduced in the abstract LC memory model grows arbitrarily large as
execution proceeds. Nevertheless, we feel that this is an interesting idea to investigate, using the results of previous work on tailoring ASM to model checking [Castillo and Winter 00, Winter 00].

We feel that an important aspect of our work is the way in which it bridges the high-level and the low-level models. The flexibility of ASM allows us to represent the LC memory model in its full generality and then move seamlessly to a model of the LC cache protocol. There have been some other attempts to bridge this gap. Shen and Arvind [Shen and Arvind 97] propose a solution based on term-rewriting rules. We believe that ASM permits more concise, readable and scalable models. Term-rewriting rules become particularly large and cumbersome when the level of abstraction is low; this problem is avoided in ASM by representing each state transformation as a (typically small) collection of local updates rather than as a single large global one. Akhiani et al. [Akhiani et al. 99] describe the use of Lamport’s Temporal Logic of Actions (TLA) [Lamport 94] in the verification of a cache coherence protocol developed by Compaq. Simply providing a specification of a full implementation of such a protocol is an impressive feat; nevertheless, it should be noted that they had difficulties in providing a full verification of the protocol and ultimately proved only a fraction of the total invariant. Of course, our verification involves only an abstract cache protocol; it remains to be seen whether our approach can handle the complexity of a true implementation. It would be interesting to test Börger’s claim [Börger 95] that the separation of verification from specification afforded by ASM can provide a simpler verification process than the integrated specification-verification system of TLA.\footnote{The specification of another cache coherence protocol was in progress at the time of the report; final results are not given.}

8 Conclusion

In this paper, we have presented formal specifications for the LC memory model and cache protocol. These specifications, contrary to the descriptions presented in [Gao and Sarkar 94] or [Gao and Sarkar 00], have been expressed rigorously. Using these formal specifications and the notions of sequential and distributed runs, we have then been able to show that the protocol indeed satisfies the model. In other words, we have shown that, using the LC protocol, any value returned by a read operation is a value legal according to the LC memory model. In addition, we also showed that the protocol is stronger than the abstract memory model: certain values that can be read by the abstract memory model cannot be read by the protocol.

An interesting path of further study for us lies in the use of model checking techniques to automate portions of our proof. Another idea we plan to pursue is to express other weak memory models in ASM terms and see how these models, and their associated protocols, compare with LC. Finally, the new memory model for the Java language proposed by Manson and Pugh [Manson and Pugh 01] uses a variant of LC; we are currently working on an ASM specification of this memory model.

\footnote{See Gurevich and Huggins [Gurevich and Huggins 97] for a critique of TLA from an ASM perspective.}
Acknowledgments

This research was partially funded by grants from the Natural Sciences and Engineering Research Council of Canada (NSERC). We thank Guang R. Gao for many valuable and lively discussions on memory models.

References


