

The Location Consistency Memory Model and Cache Protocol: Specification and Verification

Charles Wallace¹, Guy Tremblay², and José N. Amaral³

¹ Computer Science Dept., Michigan Technological University, Houghton, MI, USA

² Dépt. d'informatique, Université du Québec à Montréal, Montréal, QC, Canada

³ Computing Science Dept., University of Alberta, Edmonton, AB, Canada

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.

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* [1, 6]. 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) [6]. 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in a shared memory machine with caches requires a *cache protocol*, that invalidates or updates cached values when they no longer represent legal readable values.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) [13], 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.

In the SC model, at any time there is a unique most recent write to a location, and any other values stored in the system for that location are not legally readable; they must either be 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* [10], *lazy release consistency* [12], *entry consistency* [2], *DAG consistency* [3], and *commit, reconcile and fences* (CRF) [14]. 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* [8], 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 [9], 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 [11]. 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, *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 *strictly stronger* than the LC memory model, *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 §2 we define the common portions of the two models, producing a model LC_0 . In §3 we refine this model to arrive at a model LC_{mm} of the LC memory model. In §4 we make different refinements, resulting in a model LC_{cp} of the LC cache protocol. In §5 we prove that LC_{cp} is an implementation of LC_{mm} . In §6, we prove that LC_{cp} is in fact strictly stronger than LC_{mm} . We conclude in §7 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 *read* retrieves a value associated with a location, possibly storing it in some area local to the processor.
- A *write* adds a value to the set of values associated with the location. In any real systems, 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 *acquire* grants exclusive ownership of a location to a processor.¹ 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 *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”.

¹ 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.

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 §3 and the model of the LC cache protocol (LC_{cp}) in §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 ℓ there is a unique *agent* whose task is to perform operations on ℓ *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

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

Fig. 1. Attributes of ProcAgents.

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 may need to wait for another processor agent to release ownership of its location. When the current operation is completed, the processor agent updates its **opType** attribute.

The type of the next operation that the agent is to perform is given by the attribute **nextOpType**, an *external* (a.k.a. *monitored*) function, whose interpretation is determined by the environment. In contrast,

the attribute `opType` is updated by agents (and never by the environment), so it is called a *dynamic internal* function (or *controlled*) function.

The attribute `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 `waiting?` attribute to **true**. Finally, the monitored function `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 Fig. 1, we present the attributes for the processor agents with their *profile*.

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 **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 `initialized?` attribute that is set to **true** once the appropriate **InitAgent** has completed.

LC_0 : Ownership Agents

Function	Profile
<code>o.loc</code>	<code>OwnAgent</code> \rightarrow Location
<code>l.owner</code>	Location \rightarrow <code>ProcAgent</code>
<code>l.nextOwner</code>	Location \rightarrow <code>ProcAgent</code>

Fig. 2. Attributes of `OwnAgents` and Locations.

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 **OwnAgent** universe whose members have the responsibility of arbitrating ownership of locations and we associate a unique **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 Fig. 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.)

LC_0 : Conditions on Runs

Some aspects of our models LC_0 , LC_{mm} and LC_{cp} 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 $p.loc = \ell$.*

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

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

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:

Init condition 1 *For every Location ℓ , $\ell.owner.undef?$ 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 ℓ , 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 Fig. 3, 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. Note that these rules are redefined in §3, giving us a complete ASM model LC_{mm} . A different set of definitions for these same rules then appears in §4, resulting in a distinct ASM model LC_{cp} .

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

rule Read:
  Get next operation

rule Write:
  Get next operation

rule Acquire:
if Self.loc.owner  $\neq$  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

```

Fig. 3. Module for processor agents (ProcAgent).

LC_0 : Ownership Agent Module

```

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

```

Fig. 4. Module for ownership agents (OwnAgent).

Figure 4 contains the module for ownership agents. 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_{mm})

and the cache protocol model (LC_{cp}). In this section we present a complete specification for the LC model, LC_{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 [9], 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 \prec . Each processor must act as if it observed the events in an order compatible with \prec . When a processor performs an operation, an event is added to the history, and \prec 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 \prec 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 ℓ is ordered so that it succeeds any event already issued by p on ℓ . In other words, the set of events by p on ℓ is linearly ordered. Furthermore, since the history is a partial order and \prec is transitive, the new event also succeeds any event $e' \prec e$, 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 ℓ (by any processor agent)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.

- Which values are associated with a new read event?

In LC, when a processor agent p issues a read on a location ℓ , any write event on ℓ 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.

LC_{mm} : Universes, Attributes, and Relations

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

² Note that there is at most one latest release for ℓ at any given time, since (as pointed out in §2 and as formalized by Run Condition 3) a processor agent only releases ℓ if it has (exclusive) ownership of ℓ . Moreover, for each location the initializer agent issues a release.

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 .

Function	Profile
$e.\text{issuer}$	$\text{Event} \rightarrow \text{ProcAgent} \cup \text{InitAgent}$
$w.\text{val}$	$\text{WriteEvent} \rightarrow \text{Value}$
$p.\text{latestEvent}$	$\text{ProcAgent} \rightarrow \text{Event}$
$\ell.\text{latestRelease}$	$\text{Location} \rightarrow \text{ReleaseEvent}$
$i.\text{initWrite}$	$\text{InitAgent} \rightarrow \text{WriteEvent}$
$\text{reads?}(rd, v)$	$\text{ReadEvent} \times \text{Value} \rightarrow \text{Boolean}$
$e \prec e'$	$\text{Event} \times \text{Event} \rightarrow \text{Boolean}$

Fig. 5. Additional attributes and relations for LC_{mm} .

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

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

- $\text{reads?}(rd, v)$ indicates whether value v can be read at **ReadEvent** rd . The set of values that can be read by rd is thus $\{v \mid \text{reads?}(rd, v)\}$.
- $e \prec e'$ represents the partial order among memory events.

Attributes and relations associated with events and with locations are presented in Fig. 5.

Terminology

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

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

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

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

LC_{mm} : Terms and Transition Rules

The rules for the non-read operations by processor agents in LC_{mm} are given in Fig. 6, 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 Fig. 7. The term $\text{readOK?}(w, p)$, also defined in Fig. 7, determines whether the write value of **WriteEvent** w is readable for **ProcAgent** p . For the value of **WriteEvent** w to be readable by processor agent p , w must be a write to the appropriate location, and there must be no **WriteEvent** w' that intervenes between w and the last event issued by p .

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:

```

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  $\neq$  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  $\prec$  e := true
  do-forall c: Event: c  $\prec$  d
    c  $\prec$  e := true

```

Fig. 6. LC_{mm} rules for write, acquire and release operations.

```

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 ( $\exists w'$ : WriteEvent) w  $\prec$  w'  $\prec$  p.latestEvent

```

Fig. 7. Rule and auxiliary term for read operation in LC_{mm} .

- The new event succeeds its issuer’s latest event (as well as all predecessors of that event).
- 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. 6–7 refine the processor agent modules of LC_0 . Along with the ownership agent module in Fig. 4, they complete LC_{mm} , the 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 (abstract) assumptions about how values can be stored. In particular, we assume that each processor is equipped with its own cache, and that there is a set of memory areas collectively called *main memory*, distinct from any processor’s cache. Each location has a unique value stored in main memory. Processors store values for reading and writing in their caches. 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*. Such a 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 [9] indicate the latter. This implies that a releasing processor agent has to wait for a writeback to complete before proceeding to the next operation [9].

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 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.

LC_{cp} : Attributes

In LC_{cp} , 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

Function	Profile
$p.\text{cacheVal}$	$\text{ProcAgent} \rightarrow \text{Value}$
$p.\text{cacheValid?}$	$\text{ProcAgent} \rightarrow \text{Boolean}$
$p.\text{cacheDirty?}$	$\text{ProcAgent} \rightarrow \text{Boolean}$
$p.\text{ejectee}$	$\text{ProcAgent} \rightarrow \text{ProcAgent}$
$p.\text{latestWB}$	$\text{ProcAgent} \rightarrow \text{WritebackAgent}$
$\ell.\text{MMVal}$	$\text{Location} \rightarrow \text{Value}$
$wb.\text{issuer}$	$\text{WritebackAgent} \rightarrow \text{ProcAgent}$
$wb.\text{val}$	$\text{WritebackAgent} \rightarrow \text{Value}$
$wb.\text{active?}$	$\text{WritebackAgent} \rightarrow \text{Boolean}$

Fig. 8. Additional attributes for LC_{cp} .

another processor agent for the same processor which also has a cache entry; the cache entry of $p.\text{ejectee}$ 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 $\text{ProcAgent } p$ 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. Figure 8 summarizes the attributes used to model caches, writeback agents, and the main memory.

Terminology

In LC_{cp} , releases generally 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 PR_p at which $p.\text{opType} = \text{release}$ and ($p.\text{cacheDirty?}$ or $p.\text{wb.active?}$), we say that p prepares to release at PR_p .*

Definition *If a ProcAgent p makes a move R_p at which $p.\text{opType} = \text{release}$ and not ($p.\text{cacheDirty?}$ or $p.\text{wb.active?}$), we say that p releases at R_p .*

We use the following terms to characterize read actions and cache maintenance actions in ρ_{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 not $p.\text{latestWB.active?}$, we say that p performs a miss read at Rd_p ;*
- *Otherwise, if $p.\text{cacheDirty?}$ or $p.\text{latestWB.active?}$, 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 .⁴*

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).

- If at a move C_p , $wb_p.active?$ 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 .

LC_{cp} : 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 $p.ejectee.opType \neq \text{read}$ and $p.ejectee.opType \neq \text{write}$.

LC_{cp} : 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  $wb_p$ 
     $wb_p.val := p.cacheVal$ 
     $wb_p.active? := \text{true}$ 
     $p.latestWB := wb_p$ 

module WritebackAgent:
  if Self.active? then
    Self.loc.MMVal := Self.val
    Self.active? := false

term  $p.allWritebacksCompleted?$ :
  ( $\forall wb_p: \text{WritebackAgent}: wb_p.proc = p$ ) not  $wb_p.active?$ 

```

Fig. 9. LC_{cp} rules for cache maintenance.

The rules and terms associated with cache ejection and writeback are presented in Fig. 9. 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. The writeback agent module is simple: a writeback agent makes a single move in which it copies its value to main memory.

```

rule Read:
if not Self.cacheValid? then
  if Self.allWritebacksCompleted? then Self.cacheVal := Self.loc.MMVal
  else Self.cacheVal := Self.latestWB.val
  Self.cacheValid? := true
  if Self.ejectee.def? then Eject cache entry of Self.ejectee
Get next operation

rule Write:
Self.cacheVal := Self.writeVal
Self.cacheValid? := true
Self.cacheDirty? := true
if Self.ejectee.def? then Eject cache entry of Self.ejectee
Get next operation

rule Acquire:
if Self.loc.owner  $\neq$  Self then Self.waiting? := true
else
  if Self.cacheValid? and not Self.cacheDirty? then Self.cacheValid? := false
  Get next operation

rule Release:
if Self.cacheDirty? then Initiate writeback on cache entry of Self
elseif Self.allWritebacksCompleted? then
  Self.loc.owner := undef
  Get next operation

```

Fig. 10. 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 Fig. 10. 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.

5 LC_{cp} Obeys LC_{mm}

In this section, we outline the proof that shows that the cache protocol described by LC_{cp} implements the abstract model described by LC_{mm} . For a detailed proof, the reader should consult [15]. More precisely, our goal is to show that any value read in an execution of LC_{cp} is also a legal value in an equivalent execution of LC_{mm} . In a run of LC_{mm} , for each read operation a *set* of legal readable values is computed, while in the run of LC_{cp} a *single* value is read at each read operation. We consider runs of LC_{mm} and LC_{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_{cp} , the single value read is in the set of readable values computed at the corresponding move of LC_{mm} 's run.

Equivalent Runs of LC_{mm} and LC_{cp}

We must first start by considering what it means for runs of LC_{mm} and LC_{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 `nextOpType`, `writeVal`, and `nextOwner`.

We formalize the above as follows:

- Let σ be a state of LC_{mm} or LC_{cp} . Then σ^- is the reduct of σ to the static and monitored functions common to LC_{mm} and LC_{cp} (*i.e.*, the static and monitored functions of LC_0 , introduced in §2).
- A state σ_{mm} of LC_{mm} is *equivalent* to a state σ_{cp} of LC_{cp} if σ_{mm}^- and σ_{cp}^- are isomorphic.

Let $\rho_{cp}^* = (\mu_{cp}^*, \alpha_{cp}, \sigma_{cp}^*)$ be a run of LC_{cp} . μ_{cp}^* is a partially ordered set of moves, α_{cp} is a function mapping moves to agents (*i.e.*, gives the agent performing a move), and σ_{cp}^* is a function mapping finite initial segments of moves to states of LC_{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 [11], we lose no generality by proving correctness of an arbitrary *linearization* of a run. Hence, we can consider a linearization $\rho_{cp} = (\mu_{cp}, \alpha_{cp}, \sigma_{cp})$ of ρ_{cp}^* . μ_{cp} is a linearly ordered set of moves (an arbitrary interleaving) that has the same moves as μ_{cp}^* and preserves all the ordering of μ_{cp}^* . Since μ_{cp} is a sequence, every finite initial segment of μ_{cp} is a prefix of μ_{cp}^* , so σ_{cp} is a restriction of σ_{cp}^* to finite prefixes of μ_{cp}^* .

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

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

Lemmata: Ordering of Events in ρ_{mm}

The proof that LC_{cp} implements LC_{mm} hinges on two important properties of the ordering of events in ρ_{mm} , stated in the following lemmata. Lemma 1 states that the events issued by a `ProcAgent` are linearly ordered by $<$; 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 ρ_{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 $d_p < e_p$.*

The next property concerns how events issued by different agents in ρ_{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 ρ_{mm} , let p and q be distinct ProcAgents or for which $p.\text{loc} = q.\text{loc}$. Let W_p be a move at which p issues a WriteEvent w_p , and let Rd_q be a move after W_p at which q issues a ReadEvent. Then $w_p \prec q.\text{latestEvent}$ at Rd_q if and only if*

- p issues a ReleaseEvent r_p at a move R_p in the interval (W_p, Rd_q) and
- q issues an AcquireEvent a_q at a move A_q in the interval (R_p, Rd_q) .

Lemmata: Properties of Read Operations

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

Lemma 3 *In ρ_{cp} , let Rd_p^D be a move at which a ProcAgent p performs a dirty read of a Value v . Then in ρ_{mm} , p also reads v at Rd_p^D .*

Proof outline. An examination of ρ_{cp} shows that p reads the last value it wrote, either by consulting the cache or its last writeback agent. By Lemma 2, this write is unreadable by p in ρ_{mm} if and only if: (1) p releases its location ℓ ; (2) some other processor agent acquires ℓ , writes, and then releases; and then (3) p acquires. However, a further examination of ρ_{cp} shows that p does not release. Therefore, v is a readable value in ρ_{mm} . \square

Lemma 4 *In ρ_{cp} , let Rd_p^M be a move at which a ProcAgent p performs a miss read of a Value v . Then in ρ_{mm} , p also reads v at Rd_p^M .*

Proof outline. Let ℓ be the location associated with p . An examination of ρ_{cp} shows that p reads the last value of ℓ written back to main memory (by some q).

If $p = q$, the write is unreadable in ρ_{mm} only if p performs another write before the read. But an examination of ρ_{cp} shows that p does not do this, so the write is readable in ρ_{mm} .

If $p \neq q$, Lemma 2 implies that the write is unreadable in ρ_{mm} if and only if: (1) q releases and then (2) p acquires, and one of the following happens: (3a) q writes again before releasing; (3b) p writes after acquiring; or (3c) some other processor agent acquires ℓ , writes, and then releases. But an examination of ρ_{cp} shows that this does not happen. Therefore, v is a readable value in ρ_{mm} . \square

Lemma 5 *In ρ_{cp} , let Rd_p^C be a move at which a ProcAgent p performs a clean read of a Value v . Then in ρ_{mm} , p also reads v at Rd_p^C .*

Proof outline. An examination of ρ_{cp} shows that p reads a value from the cache, and this value was placed there as a result of either (A) a miss read (of a value written by some q) or (B) the last write by p .

In case (A), Lemma 2 implies that the write is unreadable in ρ_{mm} if and only if: (1) q releases the location ℓ and then (2) p acquires, and one of the following happens: (3a) q writes again before releasing; (3b) p writes after acquiring; or (3c) some other processor agent acquires ℓ , writes, and then releases. But a further examination of ρ_{cp} shows that this does not happen.

In case (B), Lemma 2 implies that the write is unreadable in ρ_{mm} if and only if: (1) p releases and then (2) p acquires, and in the interim (3) some other processor agent acquires ℓ , writes, and then releases. But an examination of ρ_{cp} shows that p neither releases or acquires in ρ_{cp} .

In either case, v is a readable value in ρ_{mm} . \square

Theorem: LC_{cp} Obeys LC_{mm}

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

Theorem 1 *Let Rd_p be a move of ρ_{cp} at which a ProcAgent p reads a Value v . Then at Rd_p in ρ_{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 ρ_{mm} of LC_{mm} with the following properties. In ρ_{mm} , two distinct ProcAgents p and q operate on a common Location ℓ , and no other ProcAgents operate on ℓ . (Other ProcAgents may perform operations on Locations other than ℓ .) 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 .
 Rd_q : Read by q .

At Rd_q , the value 1 is readable according to LC_{mm} , that is, in ρ_{mm} , q can read the value 1 at move Rd_q . The value 1 is unreadable at Rd_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 Rd_q . There are two cases:

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 .

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 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 [8] or [9], 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 can also show 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.

Currently, the ASM methodology has little support for automated verification compared to other techniques. Model checking of ASM specification is an active area of research [4]. An interesting area of further study for us would be to explore the use of model checking techniques to automate portions of our proof. Another area of future research is to express other weak memory models and see how these models, and their associated protocols, differ/compare with our specification of the LC memory model.

References

1. S.V. Adve and K. Gharachorloo (1995). Shared memory consistency models: a tutorial. Research Report 95/7, Digital Western Research Laboratory.
2. B. Bershad, M. Zekauskas and W. Sawdon (1993). The Midway distributed shared memory system. in *Proceedings of the IEEE COMPCON*.
3. R.D. Blumofe, M. Frigo, C.F. Joerg, C.E. Leiserson and K.H. Randall (1996). An analysis of DAG-consistent distributed shared-memory algorithms. In *Proceedings of the 8th Annual ACM Symposium on Parallel Algorithms and Architectures*, 297–308.
4. G. Del Castillo and K. Winter (2000). Model checking support for the ASM high-level language. In *Proceedings of TACAS 2000*, S. Graf and M. Schwartzbach (editors), LNCS 1785, Springer-Verlag, 331–346.
5. S. Cook and R. Reckhow (1973). Time bounded random access machines. *Journal of Computer and System Sciences* 7, 354–375.
6. D.E. Culler and J.P. Singh, with A. Gupta (1999). *Parallel computer architecture: a hardware/software approach*. Morgan Kaufmann.
7. G.R. Gao. Personal communication.
8. G.R. Gao and V. Sarkar (1994). Location consistency: Stepping beyond the barriers of memory coherence and serializability. ACAPS Technical Memo 78, School of Computer Science, McGill University.
9. G.R. Gao and V. Sarkar. Location consistency — A new memory model and cache consistency protocol. *IEEE Trans. on Comp.*, 49(8):798–813, August 2000.
10. K. Gharachorloo, D. Lenoski, J. Laudon, P. Gibbons, A. Gupta and J. Hennessy (1990). Memory consistency and event ordering in scalable shared-memory multiprocessors. In *Proceedings of the 17th Annual International Symposium on Computer Architecture*, 15–26. Also in *Computer Architecture News* 18(2).
11. Y. Gurevich (1995). Evolving Algebras 1993: Lipari guide. In E. Börger (editor), *Specification and Validation Methods*, Oxford University Press, 9–36.
12. P. Keleher, A.L. Cox and W. Zwaenepoel (1992). Lazy release consistency for software distributed shared memory. In *Proceedings of the 19th Annual International Symposium on Computer Architecture*, 13–21. Also in *Computer Architecture News* 20(2).
13. L. Lamport (1979). How to make a multiprocessor computer that correctly executes multiprocess programs. *IEEE Transactions on Computers* C-28(9): 690–691.
14. X. Shen, Arvind and L. Rudolph (1999). Commit-reconcile & fences (CRF): a new memory model for architects and compiler writers. In *Proceedings of the 26th Annual International Symposium on Computer Architecture*, 150–161.
15. C. Wallace, G. Tremblay and J.N. Amaral (2001). *The Location Consistency memory model and cache protocol: Specification and verification*. Technical Report 01-01, Computer Science Department, Michigan Technological University.